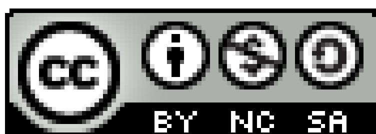


GUÍA CURSO DE INTRODUCCIÓN A LA PROGRAMACIÓN

UTN Facultad Regional Mendoza

Descripción breve

Podrá encontrar una breve introducción al mundo de la programación en Java con algunos ejemplos y ejercitación propuesta, al igual que una orientación al uso de la herramienta NetBeans.



Ing. Andrés Ceccoli
andres.ceccoli@gmail.com

Federico N. Brest
federiconbrest@gmail.com

ÍNDICE TEMÁTICO

INTRODUCCIÓN A JAVA

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una "*máquina hipotética o virtual*" denominada ***Java Virtual Machine*** (***JVM***). Era la ***JVM*** quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "*Write Once, Run Everywhere*". A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadores, ***Java*** se introdujo a finales de 1995. La clave fue la incorporación de un intérprete ***Java*** en la versión 2.0 del programa Netscape Navigator, produciendo una verdadera revolución en Internet. ***Java 1.1*** apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. ***Java 1.2***, más tarde rebautizado como ***Java 2***, nació a finales de 1998.

Al programar en ***Java*** no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de ***clases*** preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el ***API*** o ***Application Programming Interface*** de ***Java***). ***Java*** incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.). Por eso muchos expertos opinan que ***Java*** es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo del lenguaje ***Java*** es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor de ***Web***, en una base de datos o en cualquier otro lugar.
Java es un lenguaje muy completo (de hecho se está convirtiendo en un macro-lenguaje: ***Java 1.0*** tenía 12 packages; ***Java 1.1*** tenía 23 y ***Java 1.2*** tiene 59). En cierta forma casi todo depende de casi todo. Por ello, conviene aprenderlo de modo *iterativo*: primero una visión muy general, que se

va refinando en sucesivas iteraciones. Una forma de hacerlo es empezar con un ejemplo completo en el que ya aparecen algunas de las características más importantes.

La compañía ***Sun*** describe el lenguaje ***Java*** como "*simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico*". Además de una serie de halagos por parte de ***Sun*** hacia su propia criatura, el hecho es que todo ello describe bastante bien el lenguaje ***Java***, aunque en algunas de esas características el lenguaje sea todavía bastante mejorable. Algunas de las anteriores ideas se irán explicando a lo largo de este manual.

EL ENTORNO DE DESARROLLO DE JAVA

Existen distintos programas comerciales que permiten desarrollar código ***Java***. La compañía ***Sun***, creadora de ***Java***, distribuye gratuitamente el *Java(tm) Development Kit* (***JDK***). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en ***Java***. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado ***Debugger***). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la ***detección y corrección de errores***. Existe también una versión reducida del ***JDK***, denominada ***JRE*** (*Java Runtime Environment*) destinada únicamente a ejecutar código ***Java*** (no permite compilar).

Los ***IDE***s (*Integrated Development Environment*), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código ***Java***, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar ***Debug*** gráficamente, frente a la versión que incorpora el ***JDK*** basada en la utilización de una consola (denominada habitualmente ventana de comandos de MS-DOS, en ***Windows NT/95/98***) bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con ***componentes*** ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y ficheros resultantes de mayor tamaño que los basados en clases estándar.

El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el ***JDK***. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de ***Java*** (con extensión ****.java***). Si no encuentra errores en el código genera los ficheros compilados (con extensión ****.class***). En otro caso muestra la línea o líneas erróneas. En el ***JDK*** de ***Sun*** dicho compilador se llama ***javac.exe***. Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del ***JDK*** utilizada para obtener una información detallada de las distintas posibilidades.

La Java Virtual Machine

Tal y como se ha comentado al comienzo del capítulo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de ***Sun*** a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se planteó la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una "*máquina hipotética o virtual*", denominada ***Java Virtual Machine*** (***JVM***). Es esta ***JVM*** quien ***interpreta*** este código neutro convirtiéndolo a código particular de la CPU utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La ***JVM*** es el intérprete de ***Java***. Ejecuta los "***bytecodes***" (ficheros compilados con extensión ****.class***) creados por el compilador de ***Java*** (***javac.exe***). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado ***JIT*** (*Just-In-Time Compiler*), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

Instalación del entorno de desarrollo

Para poder programar en Java, necesitarás tanto el compilador (el llamado "Kit de desarrollo", JDK) como algún editor. Veremos los entornos más habituales y su instalación en Windows y Linux. En principio, el entorno más recomendable para un principiante es NetBeans, y hay alternativas aún más ligeras, como Geany, cuya instalación es inmediata en Linux pero ligeramente más incómoda en Windows.

Instalación Del JDK Y NetBeans Bajo Windows

El JDK (Java Development Kit) es la herramienta básica para crear programas usando el lenguaje Java. Es gratuito y se puede descargar desde la página oficial de Java, en el sitio web de Oracle (el actual propietario de esta tecnología, tras haber adquirido Sun, la empresa que creó Java):

www.oracle.com/technetwork/java/javase/downloads

Allí encontraremos enlaces para descargar (download) la última versión disponible.



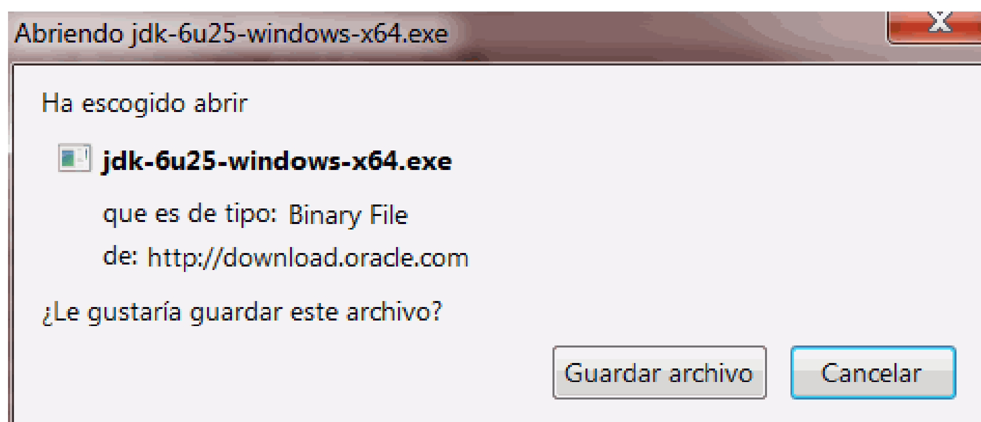
En primer lugar, deberemos escoger nuestro sistema operativo y (leer y) aceptar las condiciones de la licencia:

You must accept the Java SE 6 JDK License Agreement to download this software.

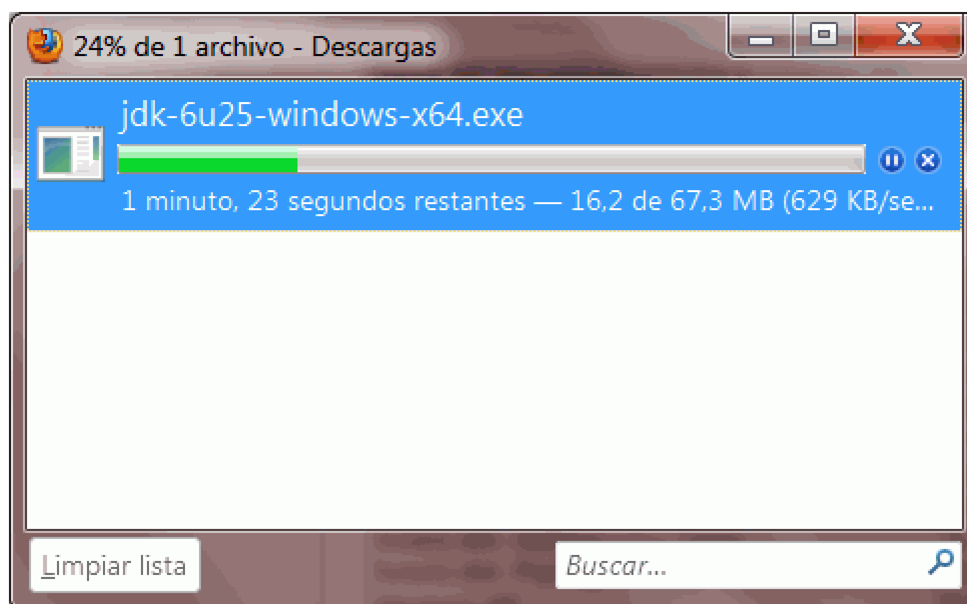
◉ Accept License Agreement ○ Decline License Agreement

Java SE Development Kit 6 Update 25		
Product / File Description	**File Size**	**Download**
Linux x86 - RPM Installer	76.85 MB	jdk-6u25-linux-i586-rpm.bin
Linux x86 - Self Extracting Installer	81.11 MB	jdk-6u25-linux-i586.bin
Linux Intel Itanium - RPM Installer	76.85 MB	jdk-6u25-linux-ia64-rpm.bin
Linux Intel Itanium - Self Extracting Installer	81.11 MB	jdk-6u25-linux-ia64.bin
Linux x64 - RPM Installer	77.06 MB	jdk-6u25-linux-x64-rpm.bin
Linux x64 - Self Extracting Installer	81.36 MB	jdk-6u25-linux-x64.bin
Solaris x86 - Self Extracting Binary	81.00 MB	jdk-6u25-solaris-i586.sh
Solaris x86 - Packages - tar.Z	136.67 MB	jdk-6u25-solaris-i586.tar.Z
Solaris SPARC - Self Extracting Binary	85.96 MB	jdk-6u25-solaris-sparc.sh
Solaris SPARC - Packages - tar.Z	141.11 MB	jdk-6u25-solaris-sparc.tar.Z
Solaris SPARC 64-bit - Self Extracting Binary	12.24 MB	jdk-6u25-solaris-sparcv9.sh
Solaris SPARC 64-bit - Packages - tar.Z	15.58 MB	jdk-6u25-solaris-sparcv9.tar.Z
Solaris x64 - Self Extracting Binary	8.49 MB	jdk-6u25-solaris-x64.sh
Solaris x64 - Packages - tar.Z	12.25 MB	jdk-6u25-solaris-x64.tar.Z
Windows x86	76.66 MB	jdk-6u25-windows-i586.exe
Windows Intel Itanium	67.27 MB	jdk-6u25-windows-ia64.exe
Windows x64	67.27 MB	jdk-6u25-windows-x64.exe

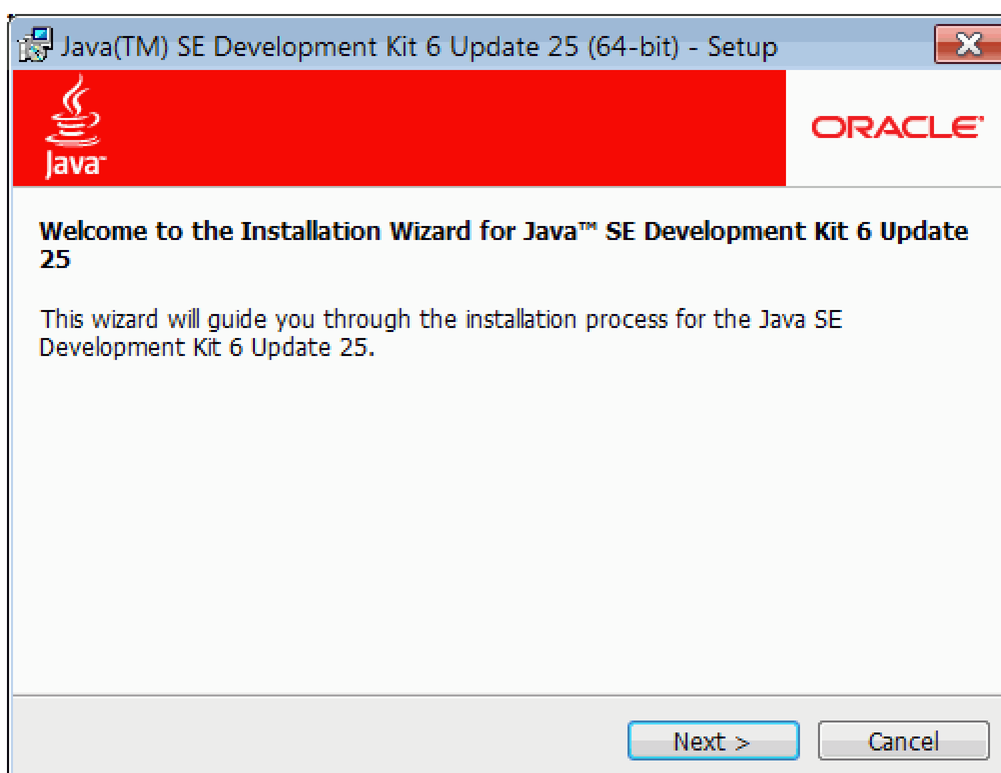
Entonces empezaremos a recibir un único fichero de gran tamaño (cerca de 70 Mb, según versiones):



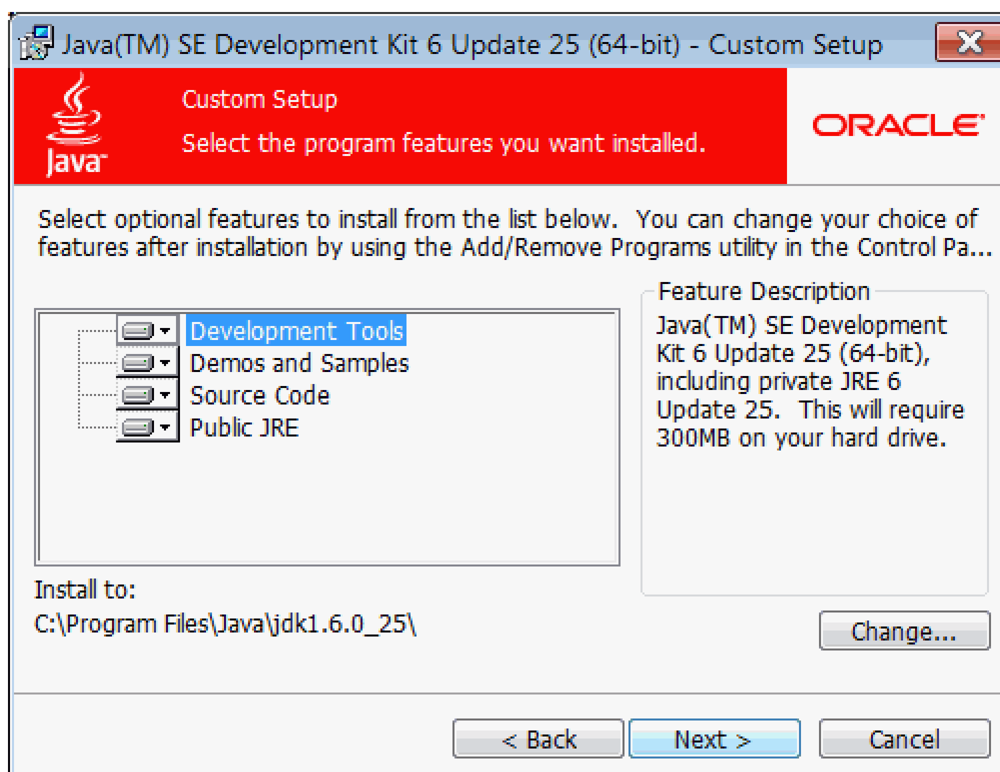
Al tratarse de un fichero de gran tamaño, la descarga puede ser lenta, dependiendo de la velocidad de nuestra conexión a Internet (y de lo saturados que estén los servidores de descarga):



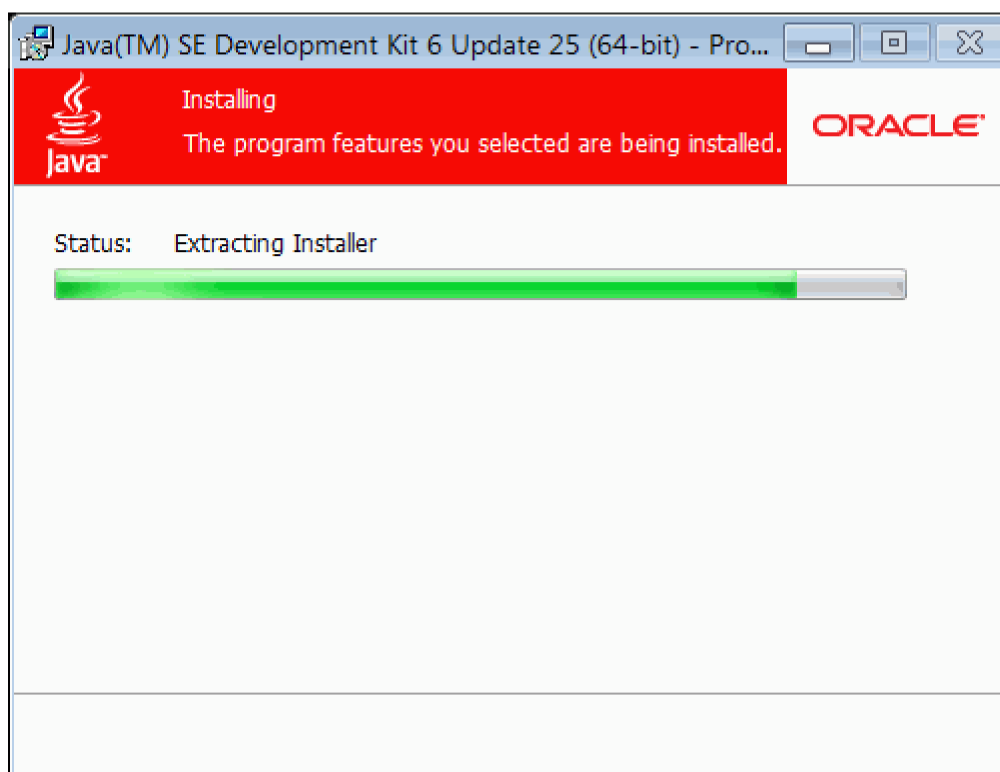
Cuando hayamos descargado, haremos doble clic en el fichero, para comenzar la instalación propiamente dicha:



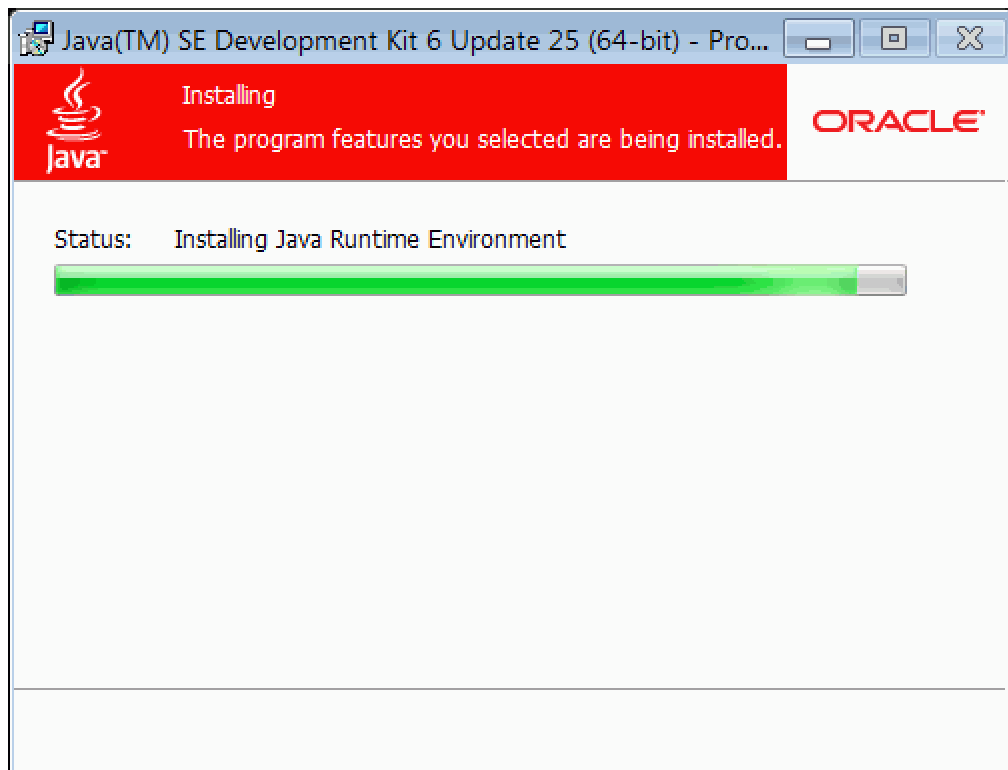
Podremos afinar detalles como la carpeta de instalación, o qué partes no queremos instalar (por ejemplo, podríamos optar por no instalar los ejemplos). Si tenemos suficiente espacio (posiblemente unos 400 Mb en total), generalmente la opción más sencilla hacer una instalación típica, sin cambiar nada:



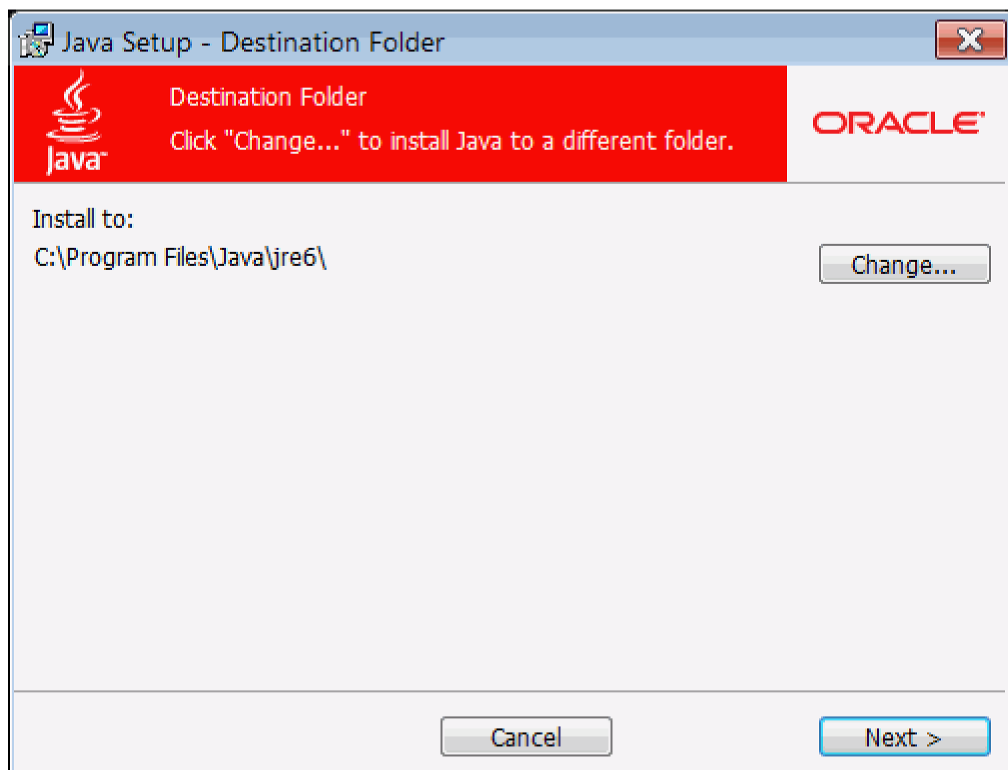
Ahora deberemos tener paciencia durante un rato, mientras se descomprime e instala todo:



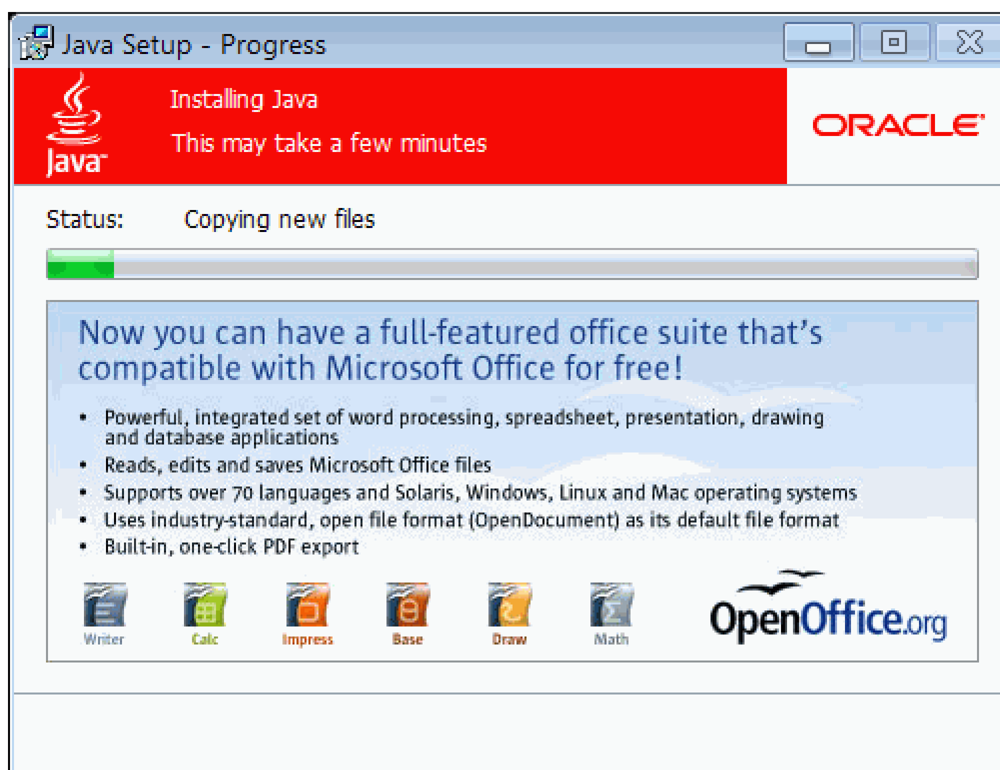
En cierto punto se nos preguntará si queremos instalar la máquina virtual Java (Java Runtime Environment, JRE). Lo razonable será responder que sí, para poder probar los programas que creemos:



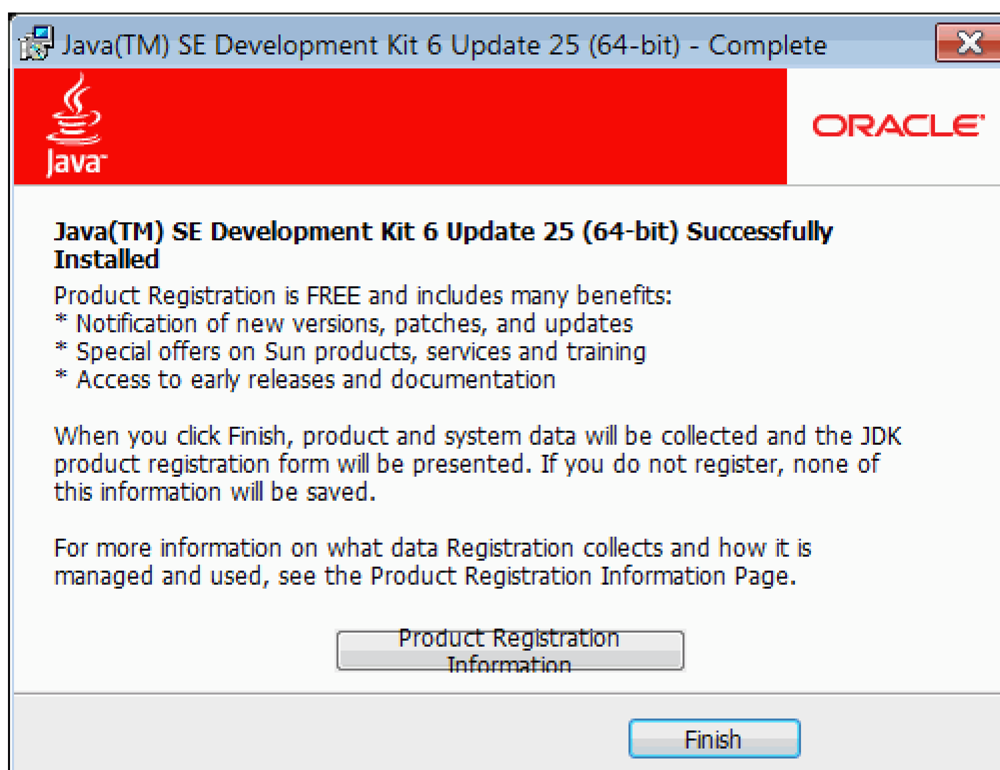
Igual que para el JDK, podríamos cambiar la carpeta de instalación:



Tendremos que esperar otro momento...



Y si todo ha ido bien, deberíamos obtener un mensaje de confirmación:



Y se nos propondrá registrar nuestra copia de Java en la página de Oracle (no es necesario):

Java Development Kit (JDK)

Registration

Please take the time to register your software.

ORACLE

You need an Oracle.com account to register your product. Create an account now, or if you already have one continue by registering your product below.

Create An Account

Need an account?

Create an Oracle.com account now.

Use My Account

Please accept the terms of use below and click "Register Now" to register your product.

I accept the terms of use for registering Oracle programs.

View terms of use

Register Now

Oracle Corporation respects your privacy. For more information on Oracle's Privacy Policy see http://www.oracle.com/html/privacy.html or contact privacy_ww@oracle.com.

ORACLE

Copyright 2010, Oracle Corporation and/or its affiliates

Con eso ya tenemos instalada la herramienta básica, el compilador que convertirá nuestros programas en Java a algo que pueda ser utilizado desde cualquier otro equipo que tenga una máquina virtual Java.

Pero el kit de desarrollo (JDK) no incluye ningún editor con el que crear nuestros programas. Podríamos instalar un "editor genérico", porque tenemos muchos gratuitos y de calidad, como Notepad++. Aun así, si nuestro equipo es razonablemente moderno, puede ser preferible instalar un entorno integrado, como NetBeans, que encontraremos en

netbeans.org

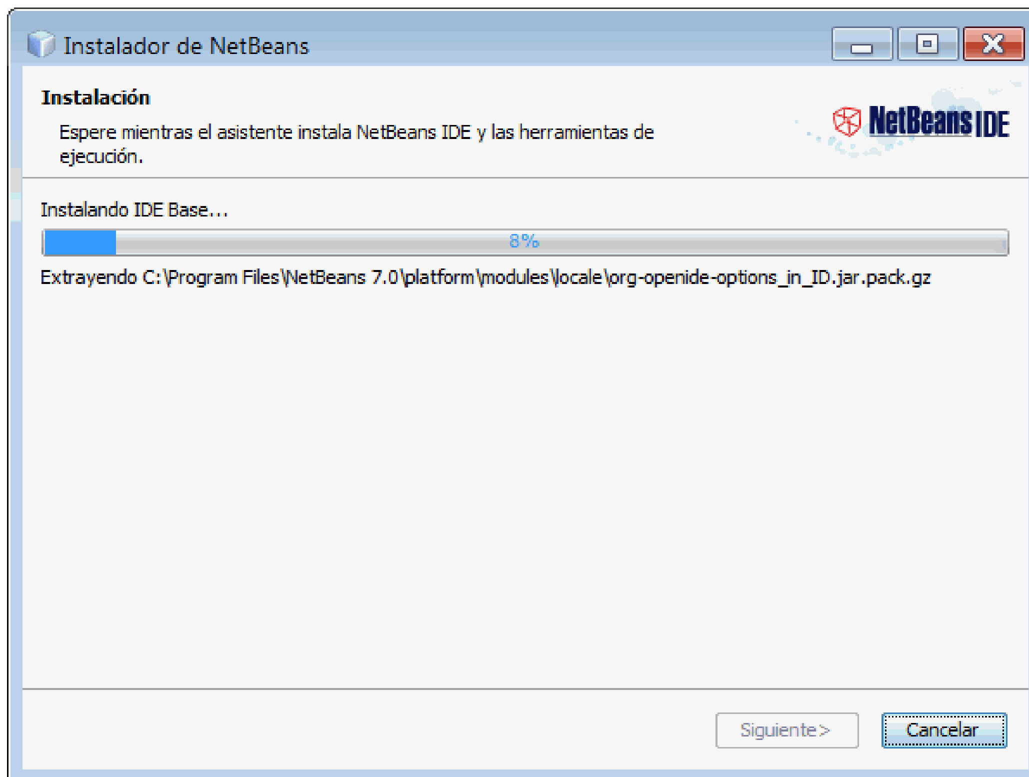


Si hacemos clic en "Download", se nos llevará a la página de descargas, en la que tenemos varias versiones para elegir. Lo razonable "para un novato" es descargar la versión para "Java SE" (Standard Edition; las alternativas son otros lenguajes, como PHP o C++, versiones profesionales como Java EE -Enterprise Edition-, o una versión que engloba todas estas posibilidades).

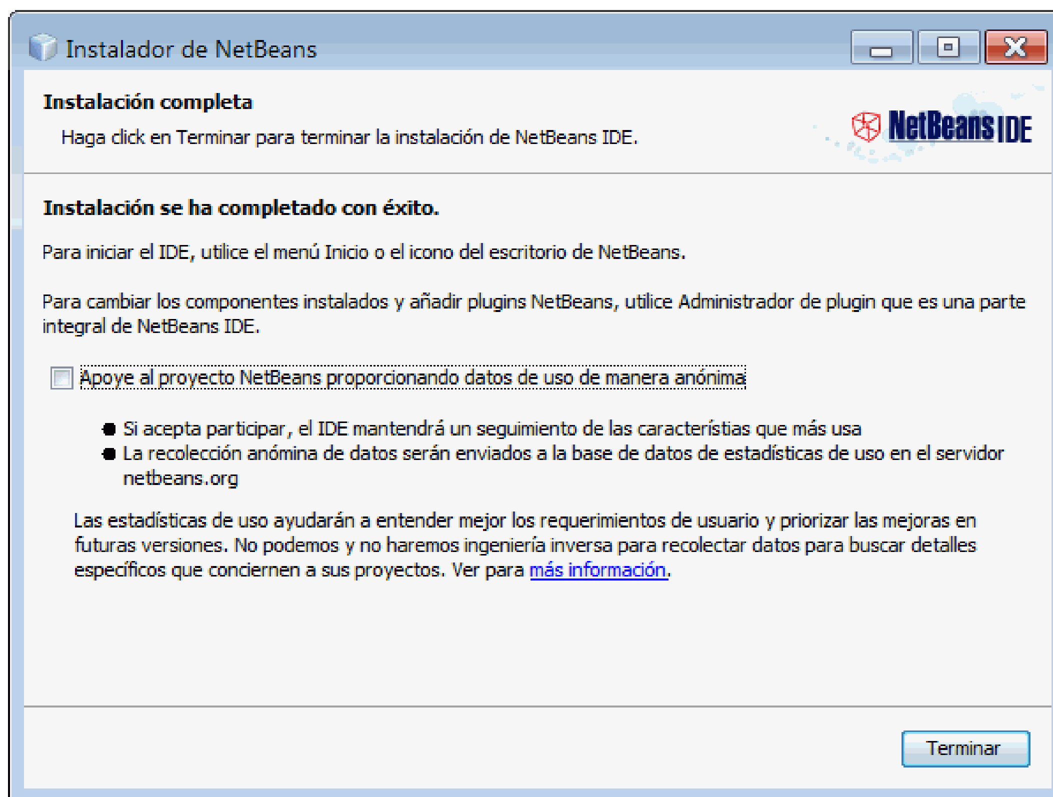
Es posible (que también podamos escoger el Español como idioma, en vez del inglés (sólo en algunas versiones).



La instalación no se podrá completar si no hemos instalado Java antes, pero si lo hemos hecho, debería ser simple y razonablemente rápida:



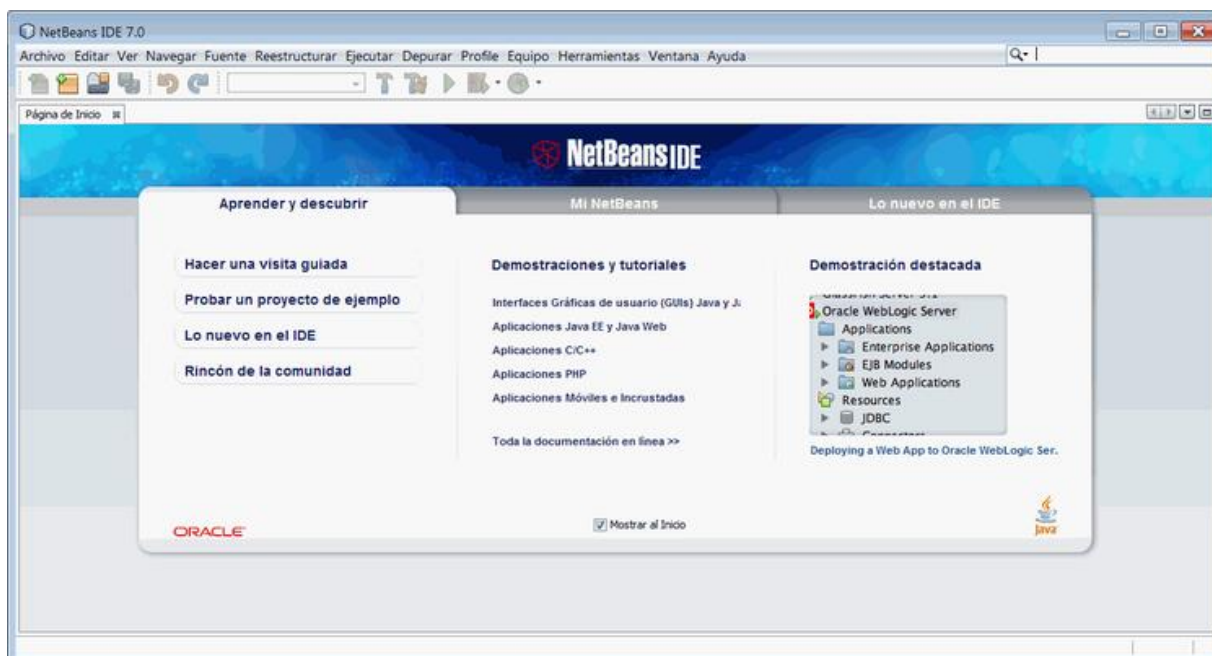
Y al final quizá se nos pregunte si queremos permitir que se recopile estadísticas sobre nuestro uso:



Todo listo. Tendremos un nuevo programa en nuestro menú de Inicio. Podemos hacer doble clic para comprobar que se ha instalado correctamente, y debería aparecer la pantalla de carga:



Y después de un instante, la pantalla "normal" de NetBeans:

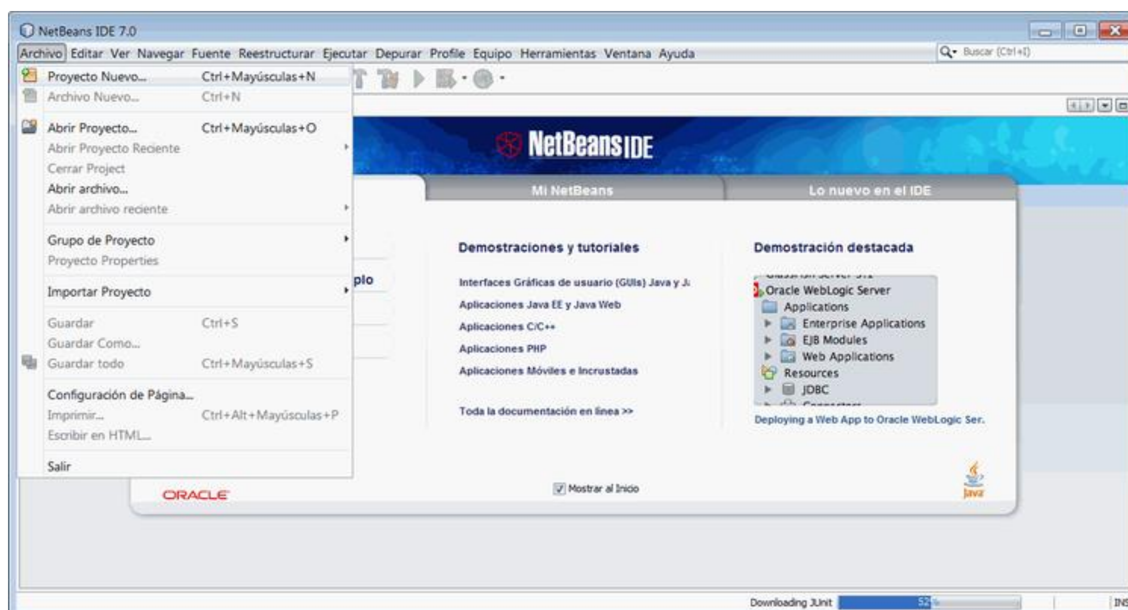


Ya estaríamos listos para empezar a crear nuestro primer programa en Java, pero eso queda para la siguiente lección...

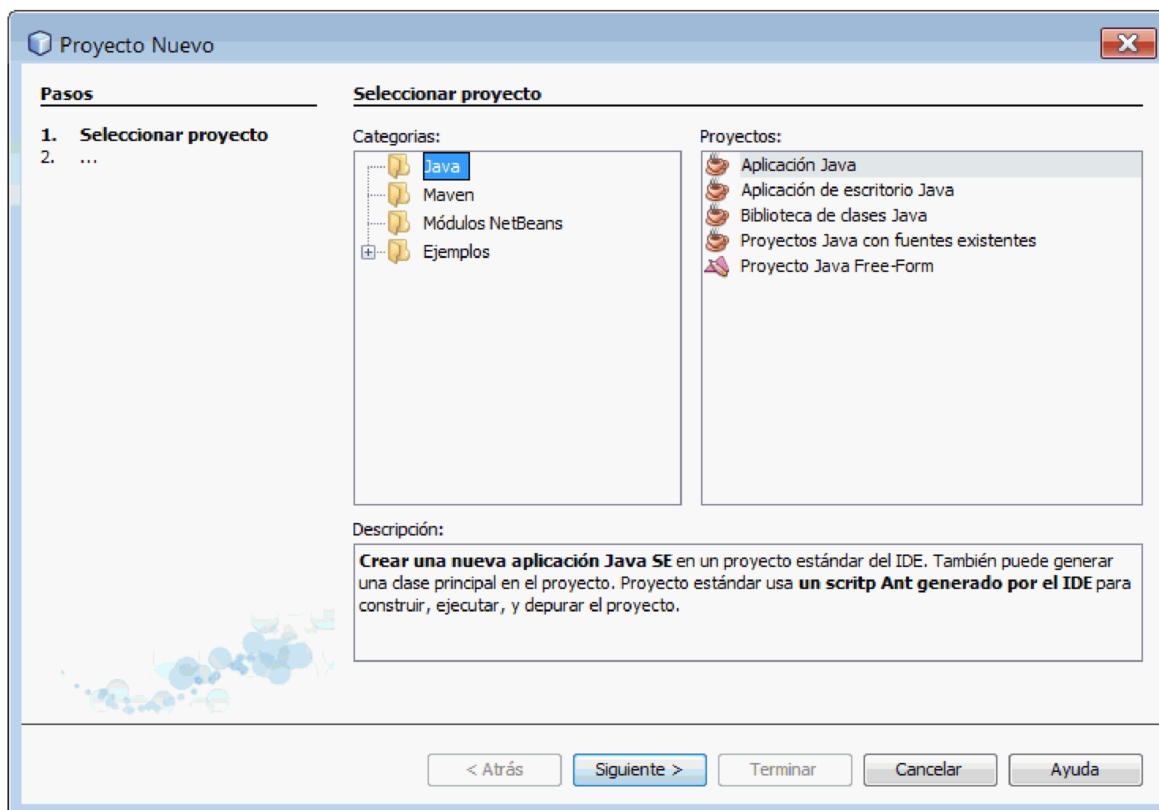
Manejo Básico De NetBeans

Vamos a ver qué pasos dar en NetBeans para crear un programa como ese.

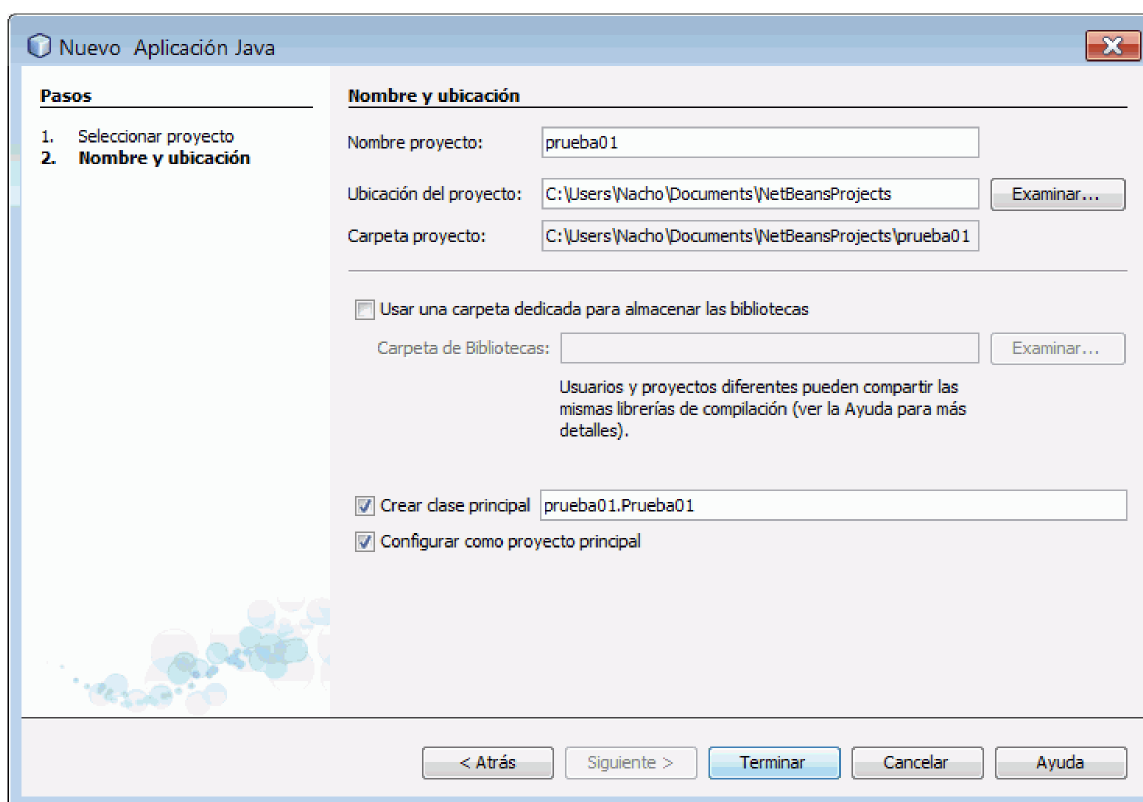
En primer lugar, deberemos entrar al menú "Archivo" y escoger la opción "Proyecto Nuevo":



Se nos preguntará el tipo de proyecto. Se tratará de una "Aplicación Java".

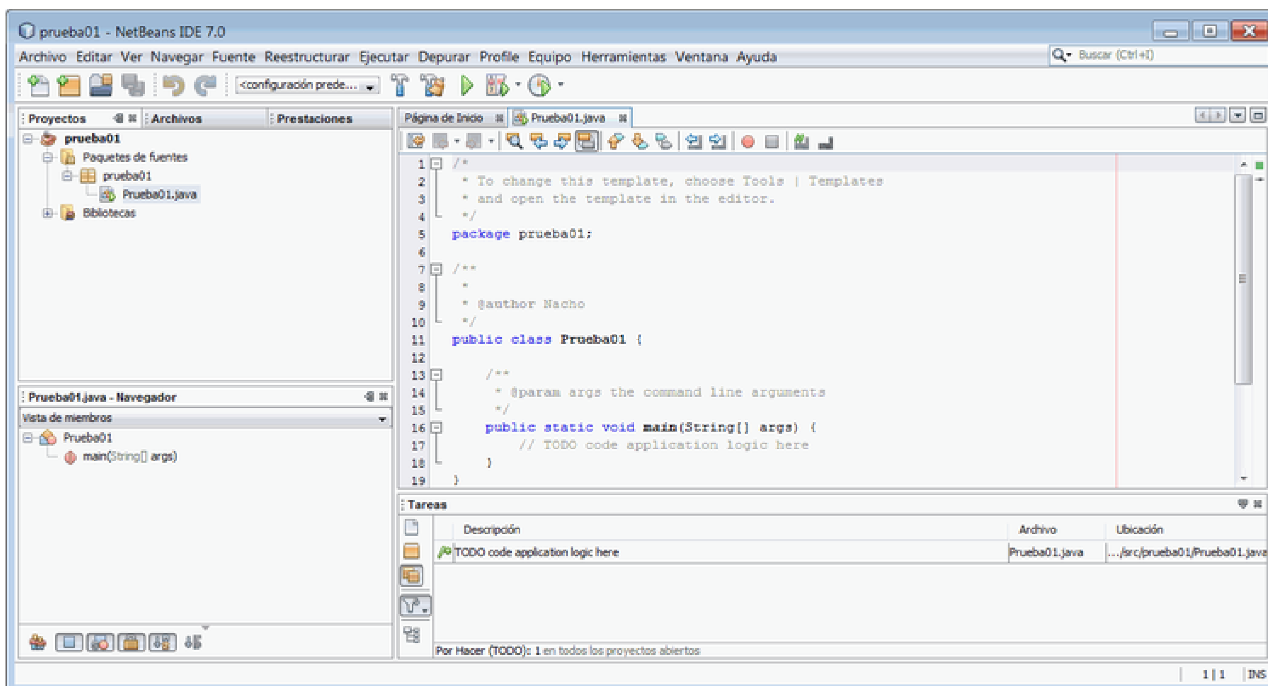


Deberemos indicar un nombre para el proyecto. Tenemos también la posibilidad de cambiar la carpeta en que se guardará.



Y entonces aparecerá un esqueleto de programa que recuerda al que nosotros queremos conseguir... salvo por un par de detalles:

- Falta la orden "System.out.print"
- Sobra una orden "package"

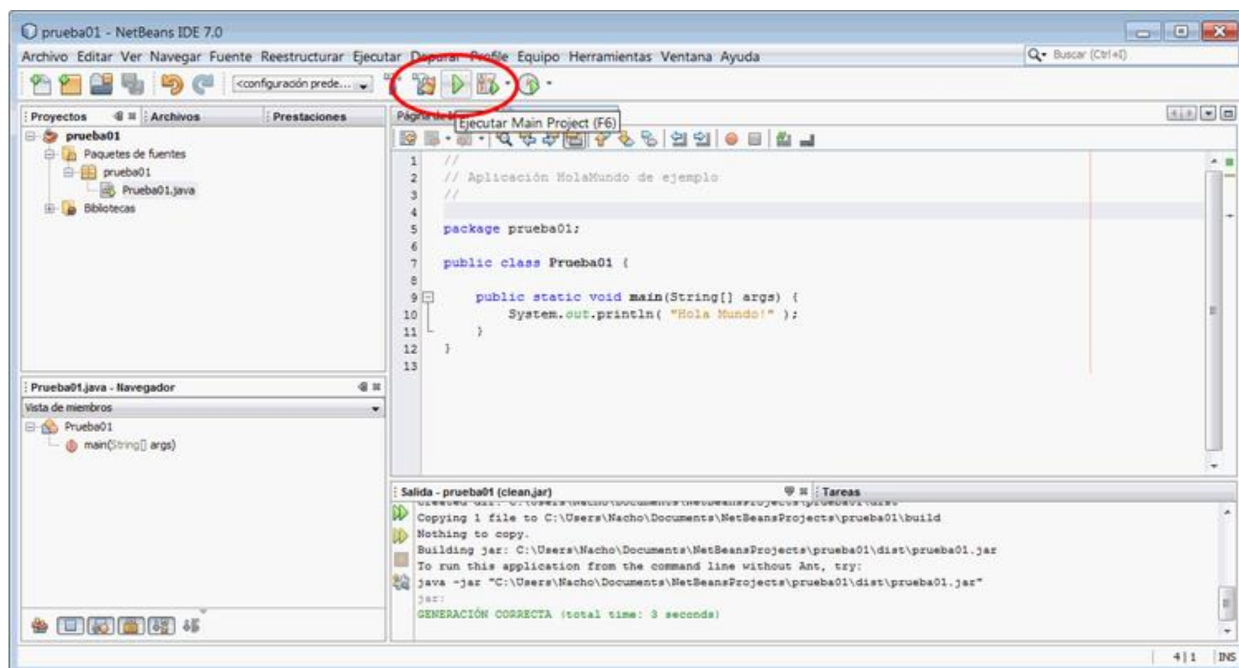


La orden "package" deberemos conservarla si usamos NetBeans, para indicar que nuestro programa es parte de un proyecto. La orden "print" deberemos añadirla, o de lo contrario nuestro programa no escribirá nada en pantalla. Podríamos borrar los comentarios adicionales, hasta llegar a algo como esto:

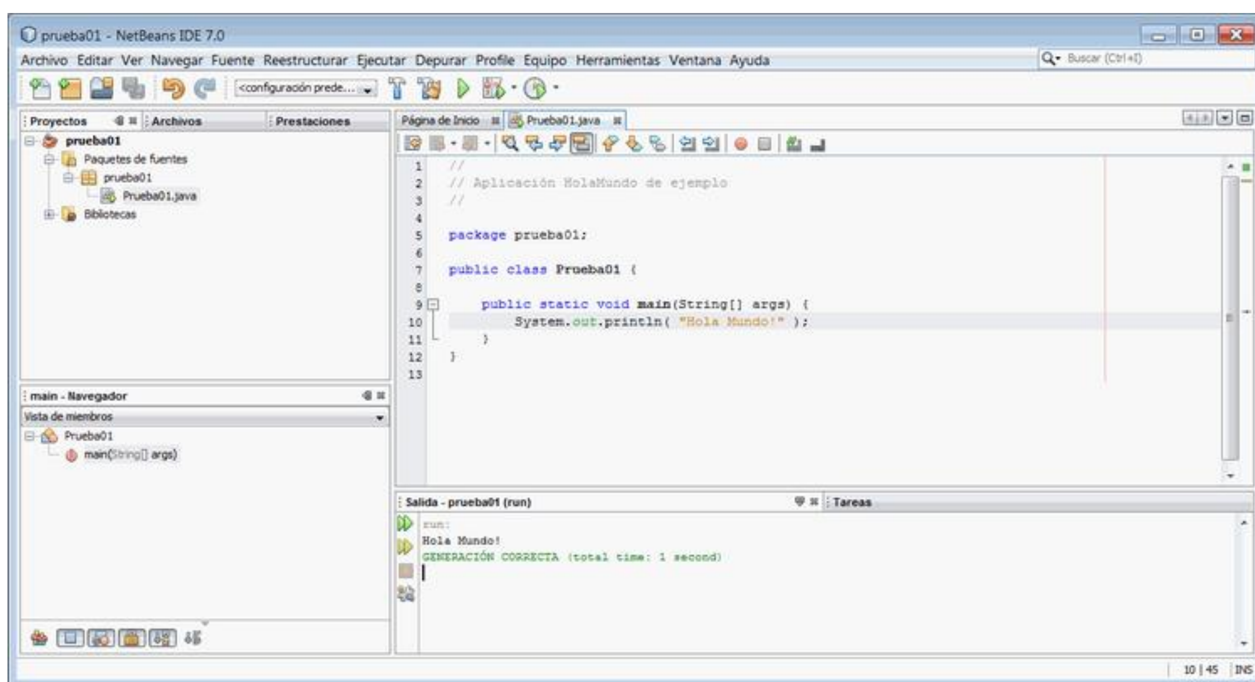
```
// HolaMundoNetBeans.java
// Aplicación HolaMundo de ejemplo, para compilar con NetBeans

package prueba01;

public class HolaMundoNetBeans {
    public static void main( String args[] ) {
        System.out.print( "Hola Mundo!" );
    }
}
```



Si hacemos clic en el botón de Ejecutar (el que muestra una punta de flecha de color verde), nuestro programa se pondrá en marcha (si no tiene ningún error), y su resultado se mostrará en la parte inferior derecha de la pantalla de trabajo de NetBeans:



Ejercicio propuesto 3.5.1: Crea un programa en Java, usando NetBeans, que te salude en pantalla por tu nombre (por ejemplo, "Hola, Nacho").

Nuestro primer programa

Un Programa Que Escribe "Hola Mundo"

Comenzaremos por crear un pequeño programa en modo texto. Este primer programa se limitará a escribir el texto "Hola Mundo!" en la pantalla. En primer lugar, veremos cómo es este programa, luego comentaremos un poco (todavía con poco detalle) las órdenes que lo forman y finalmente veremos cómo probar ese programa con distintos entornos.

Nuestro primer programa será:

```
// HolaMundo.java
// Aplicación HolaMundo de ejemplo

class HolaMundo {
    public static void main( String args[] ) {
        System.out.print( "Hola Mundo!" );
    }
}
```

Dos detalles que hay que considerar antes de seguir adelante:

- Puede parecer complicado para ser un primer programa. Es cierto, lo es. Si Java es tu primer lenguaje de programación, tienes que asumirlo: Java es así, no pretende que lo simple sea simple, sino que lo complicado no sea terrible. Por eso, los grandes proyectos serán más fáciles de mantener y menos propensos a errores que con otros lenguajes más antiguos como BASIC o C, pero los programas básicos parecerán innecesariamente complejos.
- Dentro de poco veremos cómo teclearlo. Será importante respetar las mayúsculas y minúsculas exactamente como están en el ejemplo. El hecho de escribir "system" en vez de "System" hará que obtengamos un error de compilación.

Ejercicio **propuesto 3.1.1:** Crea un programa en Java que te salude en pantalla por tu nombre (por ejemplo, "Hola, Nacho").

Entendiendo Este Primer Programa

La única parte del programa que necesitamos comprender por ahora es la línea central. Por eso, vamos a analizarlo de dentro hacia fuera, dejando todavía algunos detalles en el aire.

Escribir en pantalla

La orden que realmente escribe en pantalla es: **System.out.print("Hola Mundo!");** La orden que se encarga de escribir es "**print**". Se trata de una orden de salida (**out**) de nuestro sistema (**System**), y deberemos escribirla siempre usando esas tres palabras, una tras otra y separadas por puntos:**System.out.print**

Lo que deseemos escribir se indicará entre paréntesis. Si se trata de un texto que deba aparecer tal cual, irá además encerrado entre **comillas**.

También es importante el **punto y coma** que aparece al final de esa línea: cada orden en Java deberá terminar con punto y coma (nuestro programa ocupa varias líneas pero sólo tiene una orden, que es "print").

Por eso, la forma de escribir un texto será:

```
System.out.print( "Hola Mundo!" );
```

Formato libre

Java es un lenguaje "de formato libre". Antes de cada orden, podemos dejar tantos espacios o tantas líneas en blanco como nos apetezca. Por eso, es habitual escribir un poco más a la derecha cada vez que empecemos un nuevo bloque (habitualmente cuatro espacios), para que el programa resulte más legible.

Ese es el motivo de que nuestra orden "print" no estuviera pegada al margen izquierdo, sino más a la derecha (ocho espacios, porque está dentro de dos bloques).

```
        System.out.print( "Hola Mundo!" );
```

El cuerpo del programa

En todos los programas creados usando Java, debe existir un bloque llamado "**main**", que representa el cuerpo del programa. Por motivos que veremos más adelante, este bloque siempre comenzará con las palabras "**public static void**" y terminará con "**(String args[])**":

```
public static void main( String args[] )
```

El contenido de cada bloque del programa se debe detallar entre llaves. Por eso, la línea "print" aparece después de "main" y rodeada por llaves:

```
    public static void main( String args[] ) {
        System.out.print( "Hola Mundo!" );
    }
```

Estas llaves de comienzo y de final de un bloque se podrían escribir en cualquier punto del programa antes del contenido del bloque y después de su contenido, ya que, como hemos dicho, Java es un lenguaje de formato libre. Aun así, pro convenio, es habitual colocar la llave de apertura **al final** de la orden que abre el bloque, y la llave de cierre **justo debajo** de la palabra que abre el bloque, como se ve en el ejemplo anterior.

El nombre del programa

Cada programa Java debe "tener un nombre". Este **nombre** puede ser una única palabra o varias palabras unidas, e incluso contener cifras numéricas, pero debe empezar por una letra y no debe tener espacios intermedios en blanco. El nombre se debe indicar tras la palabra "**class**" (más adelante veremos a qué se debe el uso de esa palabra) y a continuación se abrirá el bloque que delimitará todo el programa, con llaves:

```
class HolaMundo {
    public static void main( String args[] ) {
        System.out.print( "Hola Mundo!" );
    }
}
```

El programa se debe guardar en un **fichero**, cuyo nombre coincida exactamente con lo que hemos escrito tras la palabra "class", incluso con las mismas mayúsculas y minúsculas. Este nombre de fichero terminará con ".java". Así, para nuestro primer programa de ejemplo, el fichero debería llamarse "HolaMundo.java".

Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de ***Java*** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras “//” en una línea cualquiera del

código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos /*…*/. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de cada línea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado de líneas
   ya que sólo requiere modificar
   el comienzo y el final. */
```

En ***Java*** existe además una forma especial de introducir los comentarios (utilizando /**…*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las ***clases*** y ***packages*** desarrollados por el programador. Una vez introducidos los comentarios, el programa ***javadoc.exe*** (incluido en el ***JDK***) genera de forma automática la información de forma similar a la presentada en la propia documentación del ***JDK***. La sintaxis de estos comentarios y la forma de utilizar el programa ***javadoc.exe*** se puede encontrar en la información que viene con el ***JDK***.

Varias órdenes

Si queremos "hacer varias cosas", podremos escribir varias órdenes dentro de "main". Por ejemplo, podríamos escribir primero la palabra "Hola", luego un espacio y luego "Mundo!" usando tres órdenes "print" distintas:

```
// HolaMundo2.java
// Segunda aplicación HolaMundo de ejemplo

class HolaMundo2 {
    public static void main( String args[] ) {
        System.out.print( "Hola" );
        System.out.print( " " );
        System.out.print( "Mundo!" );

    }
}
```

Como puedes imaginar, este programa deberá estar guardado en un fichero llamado "HolaMundo2.java". Como también podrás intuir, el formato libre se refiere a las órdenes del

lenguaje Java, no a los mensajes en pantalla: si escribes varios espacios con una orden "print", todos ellos serán visibles en pantalla.

Escribir y avanzar de línea

En ocasiones no querremos escribir todo en la misma línea. Para avanzar de línea tras escribir algo, basta con usar "println" en vez de "print":

```
// HolaMundo3.java
// Tercera aplicación HolaMundo de ejemplo

class HolaMundo3 {
    public static void main( String args[] ) {
        System.out.println( "Hola..." );
        System.out.println( "Mundo!" );

    }
}
```

Ejercicio propuesto 3.2.1: Crea un programa en Java que te salude en pantalla por tu nombre. En la primera línea de pantalla aparecerá la palabra "Hola" y será en la segunda línea de pantalla donde aparezca tu nombre. Usa dos órdenes "println" distintas.

CLASES

La programación orientada a objetos está basada en la definición de clases, a diferencia de la programación estructurada, que se basa en funciones. **A partir de una clase podremos crear diferentes y múltiples objetos con características similares**. La clase define atributos y métodos comunes a objetos de cierto tipo, pero cada objeto tendrá sus propias características, aunque compartan funciones entre ellos.
Antes **de poder crear un objeto**, es decir, una instancia de una clase, **debemos crear la propia clase**.
Podemos definir la estructura de una clase como:

```
class Nombre_Clase{
//Atributos
//Métodos
//Main
}
```

VARIABLES

Una ***variable*** es un ***nombre*** que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en ***Java*** hay dos tipos principales de variables:
1. Variables de ***tipos primitivos***. Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. ***Java*** permite distinta preción y distintos rangos de valores para estos tipos de variables (***char***, ***byte***, ***short***, ***int***, ***long***, ***float***, ***double***, ***boolean***). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.
2. Variables ***referencia***. Las variables referencia son referencias o nombres de una información más compleja: ***arrays*** u ***objetos*** de una determinada clase. Desde el punto de vista del papel o misión en el programa, las variables pueden ser: 1. Variables ***miembro*** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser
tipos primitivos o *referencias*. 2. Variables ***locales***: Se definen dentro de un método o más en general *dentro de cualquier*
bloque entre ***llaves*** {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también *tipos primitivos* o *referencias*.

Nombres de Variables

Los nombres de variables en ***Java*** se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por ***Java*** como operadores o separadores (,.+-*/ etc.).

Existe una serie de ***palabras reservadas*** las cuales tienen un significado especial para ***Java*** y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

abstract	boolean	break	byte	case	catch
char	class	const*	continue	default	do
double	else	extends	final	finally	float
for	goto*	if	implements	import	instanceof
int	interface	long	native	new	null
package	private	protected	public	return	short
static	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje ***Java***.

Tipos Primitivos de Variables

Se llaman ***tipos primitivos*** de variables de ***Java*** a aquellas variables sencillas que contienen los tipos de información más habituales: valores *boolean*, *caracteres* y *valores numéricos* enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores ***true*** y ***false*** (***boolean***); un tipo para almacenar caracteres (***char***), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (***byte***, ***short***, ***int*** y ***long***) y dos para valores reales de punto flotante (***float*** y ***double***). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la Tabla 2.1.

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 2.1. Tipos primitivos de variables en Java.

Los tipos primitivos de ***Java*** tienen algunas características importantes que se resumen a continuación: 1. El tipo ***boolean*** no es un valor numérico: sólo admite los valores ***true*** o ***false***. El tipo ***boolean*** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser ***boolean***. 2. El tipo ***char*** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas. 3. Los tipos ***byte***, ***short***, ***int*** y ***long*** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en ***Java*** no hay enteros ***unsigned***. 4. Los tipos ***float*** y ***double*** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente. 5. Se utiliza la palabra ***void*** para indicar la ausencia de un tipo de

variable determinado. 6. A diferencia de C/C++, los tipos de variables en ***Java*** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un ***int*** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador. 7. Existen extensiones de ***Java 1.2*** para aprovechar la arquitectura de los procesadores ***Intel***, que permiten realizar operaciones de punto flotente con una precisión extendida de 80 bits.

Cómo se definen e inicializan las variables

Una variable se define especificando el ***tipo*** y el ***nombre*** de dicha variable. Estas variables pueden ser tanto de tipos ***primitivos*** como ***referencias*** a objetos de alguna clase perteneciente al ***API*** de ***Java*** o
Tabla 2.1. Tipos primitivos de variables en Java.
ESIISS: Aprenda Java como si estuviera en Primero página 24 generada por el usuario. Si no se especifica un valor en su declaración, las variable ***primitivas*** se inicializan a cero (salvo *boolean* y *char*, que se inicializan a ***false*** y '\0'). Análogamente las variables de tipo
referencia son inicializadas por defecto a un valor especial: ***null***. Es importante distinguir entre la ***referencia*** a un objeto y el ***objeto*** mismo. Una ***referencia*** es una variable que indica dónde está guardado un objeto en la memoria del ordenador (a diferencia de C/C++, ***Java*** no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los ***punteros***). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor ***null***. Si se desea que esta ***referencia*** apunte a un nuevo objeto es necesario crear el objeto utilizando el operador ***new***. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la ***referencia*** declarada a otra referencia a un objeto existente previamente. Un tipo particular de referencias son los ***arrays*** o vectores, sean éstos de variables primitivas (por ejemplo, un vector de enteros) o de objetos. En la declaración de una referencia de tipo ***array*** hay que incluir los ***corchetes*** [ ]. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos ***MyClass***. ***Java*** garantiza que los elementos del vector son inicializados a ***null*** o a cero (según el tipo de dato) en caso de no indicar otro valor.

```
Ejemplos de declaración e inicialización de variables:
int x;              // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;          // Declaración de la variable primitiva y. Se inicializa a 5

MyClass unaRef;               // Declaración de una referencia a un objeto MyClass.
                              // Se inicializa a null
unaRef = new MyClass();       // La referencia "apunta" al nuevo objeto creado
                              // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef;  // Declaración de una referencia a un objeto MyClass.
                              // Se inicializa al mismo valor que unaRef
int [] vector;       // Declaración de un array. Se inicializa a null
vector = new int[10]; // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1};// Declaración e inicialización de un vector de 3
                              // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5];// Se crea un vector de 5 referencias a objetos
                              // Las 5 referencias son inicializadas a null
lista[0] = unaRef;            // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass();     // Se asigna a lista[1] la referencia al nuevo objeto
                              // El resto (lista[2]…lista[4] siguen con valor null
```

En el ejemplo mostrado las referencias ***unaRef***, ***segundaRef*** y ***lista[0]*** actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el ***operador igual*** (=). La forma general de las sentencias de asignación con este operador es:

variable = expression;

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La Tabla 2.2 muestra estos operadores y su equivalencia con el uso del ***operador igual*** (=).

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Tabla 2.2. Otros operadores de asignación.

Operadores unarios

Los operadores ***más*** (+) y ***menos*** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en ***Java*** es el estándar de estos operadores.

Operar Con Datos Introducidos Por El Usuario

Vamos a ver cómo hacer que sea el usuario quien introduzca valores para esas variables, que es algo mucho más habitual. Nuestro primer ejemplo sumará dos números, que en esta ocasión no estarán prefijados:

```
// Suma3.java
// Ejemplo a las variables introducidas por el usuario
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Suma3 {

    public static void main( String args[] ) {

        Scanner teclado = new Scanner(System.in);
        System.out.print( "Introduzca el primer número: " );
        int primerNumero = teclado.nextInt();
        System.out.print( "Introduzca el segundo número: " );
        int segundoNumero = teclado.nextInt();

        System.out.print( "Su suma es: " );
        System.out.println( primerNumero+segundoNumero );
    }
}
```

En este programa hay varias cosas nuevas:

- Vamos a usar una característica que no es algo básico del lenguaje. Por eso le decimos que deseamos "importar" nuevas funcionalidades. En nuestro caso, se trata de un tal "Scanner", que nos permitirá analizar textos: **import java.util.Scanner;**
- En concreto, nuestro scanner va a tomar datos desde la entrada del sistema (System.in), por lo que lo declaramos con: **Scanner teclado = new Scanner(System.in);** El nombre "teclado" podría ser "entrada" o cualquier otro.

- A partir de entonces, cada vez que llamemos a "**.nextInt()**" se leerá un número desde la entrada estándar del sistema (el teclado): int primerNumero = teclado.nextInt();

Son varias novedades, pero tampoco debería resultar difícil. Este programa escribirá algo como (dependiendo de los datos que introduzca el usuario):

```
Introduzca el primer número: 34
Introduzca el segundo número: 56
Su suma es: 90
```

Es habitual declarar las variables al principio del programa, antes de que comience la lógica real que resuelve el problema. Si varias variables van a guardar datos del mismo tipo, se pueden declarar todas ellas a la vez, separadas por comas, como en el siguiente ejemplo:

```java
// Suma3b.java
// Dos variables declaradas a la vez
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Suma3b {

    public static void main( String args[] ) {

        Scanner teclado;
        int primerNumero, segundoNumero;

        teclado = new Scanner(System.in);
        System.out.print( "Introduzca el primer número: " );
        primerNumero = teclado.nextInt();
        System.out.print( "Introduzca el segundo número: " );
        segundoNumero = teclado.nextInt();

        System.out.print( "Su suma es: " );
        System.out.println( primerNumero+segundoNumero );
    }
}
```

Además la clase Scanner proporciona otros métodos, algunos de los métodos más usados son:

METODO	DESCRIPCIÓN
nextXxx()	Devuelve el siguiente token como un tipo básico. Xxx es el tipo. Por ejemplo, nextInt() para leer un entero, nextDouble para leer un double, etc.
next()	Devuelve el siguiente token como un String.
nextLine()	Devuelve la línea entera como un String. Elimina el final \n del buffer
hasNext()	Devuelve un boolean. Indica si existe o no un siguiente token para leer.
hasNextXxx()	Devuelve un boolean. Indica si existe o no un siguiente token del tipo especificado en Xxx, por ejemplo hasNextDouble()
useDelimiter(String)	Establece un nuevo delimitador de token.

Como la forma de asimilar todo esto es probándolo, aquí tienes varios ejercicios propuestos:

Ejercicio propuesto 4.3.1: Crea un programa que calcule y muestre el producto de dos números enteros que introduzca el usuario.

Ejercicio propuesto 4.3.2: Crea un programa que calcule y muestre la división de dos números enteros introducidos por el usuario.

Ejercicio propuesto 4.3.3: Crea un programa que calcule y muestre el resto de la división de dos números enteros que introduzca el usuario.

Ejercicio propuesto 4.3.4: Crea un programa que pida al usuario una longitud en millas (por ejemplo, 3) y calcule su equivalencia en metros (1 milla = 1609 m).

Ejercicio propuesto 4.3.5: Crea un programa que pida al usuario una temperatura en grados centígrados y calcule (y muestre) a cuántos grados Fahrenheit equivalen (F = 9*C/5 + 32).

Booleanos, caracteres, cadenas de texto y arrays

Hemos visto cómo manejar datos numéricos, tanto enteros como reales, y con mayor o menor precisión, pero para muchos problemas reales necesitaremos datos de otros tipos: textos, datos estructurados e incluso otros datos simples que aún no hemos tratado, como letras individuales y

Datos Booleanos

Un dato "booleano" es uno que puede tener sólo dos valores posibles: verdadero (true) o falso (false), que son los dos valores existentes en la "lógica de Boole", de la que toman su nombre.

Es habitual utilizarlos para hacer que las condiciones complicadas resulten más legibles.

Caracteres

El tipo de datos **char** lo usaremos para almacenar una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación, o cualquier otro carácter. Ocupa 2 bytes. Sigue un estándar llamado Unicode (que a su vez engloba a otro estándar anterior llamado ASCII). Sus valores se deben indicar entre comillas simples: char inicial = 'n'.

Java no permite usar un "Scanner" para leer desde teclado letra a letra.

Las Cadenas De Texto

Una cadena de texto (en inglés, "string") es un conjunto de letras, que usaremos para poder almacenar palabras y frases. Realmente en Java hay **dos "variantes"** de las cadenas de texto: existe un tipo de datos llamado "String" (con la primera letra en mayúsculas, al contrario que los tipos de datos que habíamos visto hasta ahora) y otro tipo de datos llamado "StringBuilder". Un "**String**" será una cadena de caracteres constante, que no se podrá modificar (podremos leer su valor, extraer parte de él, etc.; pero para cualquier modificación, deberemos volcar los datos a una nueva cadena), mientras que un "**StringBuilder**" se podrá modificar "con más facilidad" (podremos insertar letras, dar la vuelta a su contenido, etc) a cambio de ser ligeramente menos eficiente (más lento).

String

Las cadenas se declaran con la palabra "**String**" (cuidado: la primera letra debe estar en mayúsculas) y su valor se asigna entre dobles comillas:

```
// String1.java
// Primer contacto con los String (cadenas de texto)
// Introducción a Java, Nacho Cabanes

class String1 {
    public static void main( String args[] ) {
```

```
        String saludo = "Hola";

        System.out.print( "El saludo es... " );
        System.out.println( saludo );

    }
}
```

Podemos "concatenar" cadenas (juntar dos cadenas para dar lugar a una nueva) con el signo +, el mismo que usamos para sumar números. Es frecuente utilizarlo para escribir varias cosas en una misma frase:

```
// String2.java
// Primer contacto con los String: concatenación
// Introducción a Java, Nacho Cabanes

class String2 {
    public static void main( String args[] ) {

        String saludo = "Hola " + "tú";
        System.out.println( "El saludo es... " + saludo );

    }
}
```

Una lista un poco más detallada de los "métodos" (operaciones con nombre) que se pueden aplicar sobre una cadena podría ser::

Método	***Cometido***
length()	Devuelve la longitud (número de caracteres) de la cadena
charAt (int pos)	Devuelve el carácter que hay en una cierta posición
toLowerCase()	Devuelve la cadena convertida a minúsculas
toUpperCase()	Devuelve la cadena convertida a mayúsculas
substring(int desde, int cuantos)	Devuelve una subcadena: varias letras a partir de una posición dada
replace(char antiguo, char nuevo)	Devuelve una cadena conun carácter reemplazado por otro
trim()	Devuelve una cadena sin espacios de blanco iniciales ni finales

startsWith(String subcadena)	Indica si la cadena empieza con una cierta subcadena
endsWith(String subcadena)	Indica si la cadena termina con una cierta subcadena
indexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el principio, a partir de una posición opcional)
lastIndexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el final, a partir de una posición opcional)
valueOf(objeto)	Devuelve un String que es la representación como texto del objeto que se le indique (número, boolean, etc.)
concat(String cadena)	Devuelve la cadena con otra añadida a su final (concatenada) También se pueden concatenar cadenas con "+"
equals(String cadena)	Mira si las dos cadenas son iguales (lo mismo que "= =")
equals-IgnoreCase(String cadena)	Comprueba si dos cadenas son iguales, pero despreciando las diferencias entre mayúsculas y minúsculas
compareTo(String cadena2)	Compara una cadena con la otra (devuelve 0 si son iguales, negativo si la cadena es "menor" que cadena2 y positivo si es "mayor").

En ningún momento estamos modificando el String de partida. Eso sí, en muchos de los casos creamos un String modificado a partir del original.

Los Arrays

Imaginemos que tenemos que realizar cálculos estadísticos con 10 números que introduzca el usuario. Parece evidente que tiene que haber una solución más cómoda que definir 10 variables distintas y escribir 10 veces las instrucciones de avisar al usuario, leer los datos que teclee, y almacenar esos datos. Si necesitamos manejar 100, 1.000 o 10.000 datos, resulta todavía más claro que no es eficiente utilizar una variable para cada uno de esos datos.

Por eso se emplean los arrays (que en ocasiones también se llaman "matrices" o "vectores", por similitud con estas estructuras matemáticas, y que algunos autores traducen como "arreglos"). Un **array** es una variable que puede contener varios datos del mismo tipo. Para acceder a cada uno de esos datos, indicaremos su posición entre corchetes. Por ejemplo, si definimos una variable llamada "m" que contenga 10 números enteros, accederemos al primero de estos números como m[0], el último como m[9] y el quinto como m[4] (se empieza a numerar a desde 0 y se termina en n-1). Veamos un ejemplo que halla la media de cinco números (con decimales, "double"):

```
// Array1.java
// Aplicación de ejemplo con Arrays
// Introducción a Java, Nacho Cabanes
```

```
class Array1 {
    public static void main( String args[] ) {
        double[] a = { 10, 23.5, 15, 7, 8.9 };
        double total = 0;
        int i;

        for (i=0; i<5; i++)
            total += a[i];

        System.out.println( "La media es:" );
        System.out.println( total / 5 );
    }
}
```

Para definir la variable podemos usar **dos formatos**: "double a[]" (que es la sintaxis habitual en C y C++, no recomendada en Java) o bien "double[] a" (que es la sintaxis recomendada en Java, y posiblemente es una forma más "razonable" de escribir "la variable a es un array de doubles").

Lo habitual no será conocer los valores en el momento de teclear el programa, como hemos hecho esta vez. Será mucho más frecuente que los datos los teclee el usuario o bien que los leamos de algún fichero, los calculemos, etc. En este caso, tendremos que reservar el espacio, y los valores los almacenaremos a medida que vayamos conociéndolos.

Para ello, primero **declararíamos** que vamos a utilizar un array, así:

```
double[] datos;
```

y después **reservaríamos** espacio (por ejemplo, para 1.000 datos) con

```
datos = new double [1000];
```

Si conocemos el tamaño desde el primer momento, estos dos pasos se pueden dar en uno solo, así:

```
double[] datos = new double [1000];
```

y daríamos los valores de una forma similar a la que hemos visto en el ejemplo anterior:

```
datos[25] = 100 ; datos[0] = i*5 ; datos[j+1] = (j+5)*2;
```

Vamos a ver un ejemplo algo más completo, con tres arrays de números enteros, llamados a, b y c. A uno de ellos (a) le daremos valores al definirlo, otro lo definiremos en dos pasos (b) y le daremos fijos, y el otro lo definiremos en un paso y le daremos valores calculados a partir de ciertas operaciones:

```
// Array2.java
// Aplicación de ejemplo con Arrays
// Introducción a Java, Nacho Cabanes

class Array2 {
    public static void main( String args[] ) {

        int i; // Para repetir con bucles "for"

        // ------  Primer array de ejemplo, valores prefijados
        int[] a = { 10, 12345, -15, 0, 7 };
        System.out.print( "Los valores de a son: " );
        for (i=0; i<5; i++)
            System.out.print( a[i] + " " );
        System.out.println( );

        // ------  Segundo array de ejemplo, valores uno por uno
        int[] b;
        b = new int [3];
        b[0] = 15; b[1] = 132; b[2] = -1;
        System.out.print( "Los valores de b son: " );
        for (i=0; i<3; i++)
            System.out.print( b[i] + " " );
        System.out.println( );

        // ------  Tercer array de ejemplo, valores con "for"
        int j = 4;
        int[] c = new int[j];
        for (i=0; i < j; i++)
            c[i] = (i+1)*(i+1);
        System.out.print( "Los valores de c son: " );
```

```
        for (i=0; i < j; i++)
            System.out.print( c[i] + " " );
        System.out.println( );
    }
}
```

Operadores

Operador condicional ?:

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

booleanExpression ? res1 : res2

donde se evalúa booleanExpression y se devuelve res1 si el resultado es true y res2 si el resultado es false. Es el único operador ternario (tres argumentos) de Java. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:
x=1 ; y=10; z = (x<y)?x+3:y+8; asignarían a z el valor 4, es decir x+3.

Operadores incrementales

Java dispone del operador incremento (++) y decremento (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas: 1. Precediendo a la variable (por ejemplo: ++i). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece. 2. Siguiendo a la variable (por ejemplo: i++). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles ***for*** es una de las aplicaciones más frecuentes de estos operadores.

Operadores relacionales

Los ***operadores relacionales*** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor ***boolean*** (***true*** o ***false***) según se cumpla o no la relación considerada. La Tabla 2.3 muestra los operadores relacionales de ***Java***. Estos operadores se utilizan con mucha frecuencia en las ***bifurcaciones*** y en los ***bucles***, que se verán en próximos apartados de este capítulo.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Tabla 2.3. Operadores relacionales.

Operadores lógicos

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 2.4. Operadores lógicos.

Los operadores lógicos se utilizan para construir ***expresiones lógicas***, combinando valores lógicos (***true*** y/o ***false***) o los resultados de los operadores ***relacionales***. La Tabla 2.4 muestra los operadores lógicos de ***Java***. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser ***true*** y el primero es ***false***, ya se sabe que la condición de que ambos sean ***true*** no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia: System.out.println("El total asciende a " + result + " unidades"); donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método ***println()***. La variable numérica ***result*** es convertida automáticamente por ***Java*** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de *x/y*z* depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de ***mayor a menor*** precedencia:

postfix operators	[] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

En ***Java***, todos los operadores binarios, excepto los operadores de asignación, se evalúan de ***izquierda a derecha***. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

ESTRUCTURAS DE PROGRAMACIÓN

En este apartado se supone que el lector tiene algunos conocimientos de programación y por lo tanto no se explican en profundidad los conceptos que aparecen. Las ***estructuras de programación*** o ***estructuras de control*** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados ***bifurcaciones*** y ***bucles***. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a *concepto*, aunque su *sintaxis* varía de un lenguaje a otro. La sintaxis de ***Java*** coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

Sentencias o expresiones

Una ***expresión*** es un conjunto variables unidos por ***operadores***. Son órdenes que se le dan al computador para que realice una tarea determinada. Una ***sentencia*** es una ***expresión*** que acaba en ***punto y coma*** (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es

utilizar una línea para cada sentencia. Por ejemplo:
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias

Condicionales

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el *flujo de ejecución* de un programa. Existen dos bifurcaciones diferentes: ***if*** y ***switch***.

Condicional if

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor true). Tiene la forma siguiente:

```
if (booleanExpression) {
    statements;
}
```

Las ***llaves*** {} sirven para agrupar en un ***bloque*** las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del ***if***.

Condicional if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el ***else*** se ejecutan en el caso de no cumplirse la expresión de comparación (***false***),

```
if (booleanExpression) {
    statements1;
} else {
    statements2;
}
```

Condicional if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al ***else***.

```
if (booleanExpression1) {
    statements1;
} else if (booleanExpression2) {
    statements2;
} else if (booleanExpression3) {
    statements3;
} else {
    statements4;
}
```

Sentencia switch

Se trata de una alternativa a la bifurcación ***if elseif else*** cuando se compara la ***misma expresión*** con distintos valores. Su forma general es la siguiente:

```
switch (expression) {
    case value1:  statements1; break;
    case value2:  statements2; break;
    case value3:  statements3; break;
    case value4:  statements4; break;
    case value5:  statements5; break;
    case value6:  statements6; break;
    [default: statements7;]
}
```

Las características más relevantes de ***switch*** son las siguientes: 1. Cada sentencia ***case*** se corresponde con un único valor de ***expression***. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos. El ejemplo del Apartado 2.3.3.3 no se podría realizar utilizando ***switch***. 2. Los valores no comprendidos en ninguna sentencia ***case*** se pueden gestionar en ***default***, que es opcional. 3. En ausencia de ***break***, cuando se ejecuta una sentencia ***case*** se ejecutan también todas las ***case*** que van a continuación, hasta que se llega a un ***break*** o hasta que se termina el ***switch***.

Bucles

Un ***bucle*** se utiliza para realizar un proceso repetidas veces. Se denomina también ***lazo*** o ***loop***. El código incluido entre las ***llaves*** {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (***booleanExpression***) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

While

La orden "**while**" que una parte del programa se repita mientras se cumpla una cierta condición. Su formato será:

```
while (condición)
    sentencia;
```

Es decir, la sintaxis es similar a la de "if", con la diferencia de que aquella orden realizaba la sentencia indicada una vez como máximo (**si** se cumplía la condición), pero "while" puede repetir la sentencia más de una vez (**mientras** la condición sea cierta). Al igual que ocurría con "if", podemos realizar varias sentencias seguidas (dar "más de un paso") si las encerramos entre llaves:

```
// While1.java
// Comprobación repetitiva de condiciones con "while"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class While1 {

    public static void main( String args[] ) {

        Scanner teclado = new Scanner(System.in);
        System.out.print("Introduce un cero: ");
        int dato = teclado.nextInt();
        while (dato != 0) {
            System.out.print("No era cero. Introduce cero: ");
            dato = teclado.nextInt();
        }
        System.out.println("Terminado!");
    }
}
```

Estas estructuras repetitivas se pueden usar también para hacer "contadores". Por ejemplo, un programa que contase del 1 al 5 (y escribiese todos esos números) podría ser:

```
// Contar1a5.java
```

```
// Contar del 1 al 5 con "while"
// Introducción a Java, Nacho Cabanes

class Contar1a5 {

    public static void main( String args[] ) {

        int x = 1;

        while (x <= 5) {
            System.out.println( x );
            x++;
        }
    }
}
```

Ejercicio propuesto 6.1.1: Crea un programa que muestre los números del 1 al 10, usando "while"

Ejercicio propuesto 6.1.2: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "while"

Ejercicio propuesto 6.1.3: Crea un programa que muestre la "tabla de multiplicar del 5", usando "while"

Ejercicio propuesto 6.1.4: Crea un programa que pida al usuario un número entero y muestre su cuadrado. Se repetirá mientras el usuario introduzca un número distinto de cero.

Ejercicio propuesto 6.1.5: Crea un programa que pida al usuario su contraseña (un número entero). Se repetirá mientras introduzca una contraseña distinta de "1234".

Ejercicio propuesto 6.1.6: Crea un programa que pida al usuario su login (un número entero) y su contraseña (otro número entero). Se repetirá mientras el usuario introduzca un login distinto de "1809" o una contraseña distinta de "1234".

Ejercicio propuesto 6.1.7: Crea un programa que escriba en pantalla tantos asteriscos como el usuario indique, todos ellos en la misma línea.

Do-While

Existe una variante de "while", que permite comprobar la condición al final de la parte repetitiva, en vez de hacerlo antes de comenzar. Es el conjunto **do..while**, cuyo formato es:

```
do {
    sentencia;
} while (condición)
```

Con "while", si la condición era falsa desde un principio, los pasos que se indicaban a continuación de "while" no llegaban a darse ni una sola vez; con do-while, las "sentencias" intermedias se realizarán al menos una vez.

Un ejemplo típico de esta construcción "do..while" es cuando queremos que el usuario introduzca una contraseña que le permitirá acceder a una cierta información:

```
// DoWhile1.java
// Comprobación repetitiva de condiciones con "do-while"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class DoWhile1 {

    public static void main( String args[] ) {

        int password;

        Scanner teclado = new Scanner(System.in);

        do {

            System.out.print( "Introduzca su password numérica: " );
            password = teclado.nextInt();

            if (password != 1234)
                System.out.println( "No es válida." );
        }
        while (password != 1234);
    }
```

```
}
```

Por supuesto, también podemos crear un contador usando un do..while, siempre y cuando tengamos en cuenta que la condición se comprueba al final, lo que puede hacer que la lógica sea un poco distinta en algunos casos:

```
// Contar1a5b.java
// Contador con "do-while"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Contar1a5b {

    public static void main( String args[] ) {

        int x = 1;

        do {
            System.out.println( x );
            x++;
        }
        while (x <= 5);
    }
}
```

Ejercicio propuesto 6.2.1: Crea un programa que muestre los números del 1 al 10, usando "do-while"

Ejercicio propuesto 6.2.2: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "do-while"

Ejercicio propuesto 6.2.3: Crea un programa que muestre la "tabla de multiplicar del 5", usando "do-while"

Ejercicio propuesto 6.2.4: Crea un programa que pida al usuario dos números enteros y muestre su suma. Se repetirá hasta que los dos sean 0. Emplea do-while.

Ejercicio propuesto 6.2.5: Crea un programa que pida al usuario su login (un número entero) y su contraseña (otro número entero). Se repetirá hasta que el usuario introduzca como login "1809" y como contraseña "1234". En esta ocasión, hazlo con do-while.

Ejercicio propuesto 6.2.6: Crea un programa que pida un número de tipo byte al usuario y escriba en pantalla un cuadrado formado por asteriscos, que tendrá como alto y ancho la cantidad introducida por el usuario.

For

Una tercera forma de conseguir que parte de nuestro programa se repita es mediante la orden "**for**". La emplearemos sobre todo para conseguir un número concreto de repeticiones. En lenguajes como BASIC su formato es muy simple: "FOR x = 1 TO 10" irá recorriendo todos los valores de x, desde uno hasta 10. En Java y otros lenguajes que derivan de C, su sintaxis es más enrevesada:

```
for ( valor_inicial ; condicion_continuacion ; incremento ) {
sentencias
}
```

Es decir, indicamos entre paréntesis, y separadas por puntos y coma, tres órdenes:

- La primera orden dará el valor inicial a una variable que sirva de control.
- La segunda orden será la condición que se debe cumplir mientras que se repitan las sentencias.
- La tercera orden será la que se encargue de aumentar -o disminuir- el valor de la variable, para que cada vez quede un paso menos por dar.

Esto se verá mejor con un ejemplo. Podríamos repetir 10 veces un bloque de órdenes haciendo:

```
// For1.java
// Repetición con "for" 1: escribir 10 veces
// Introducción a Java, Nacho Cabanes

class For1 {

    public static void main( String args[] ) {

        int i;

```

```
        for ( i=1 ; i<=10 ; i++ ) {
            System.out.print( "Hola " );
        }
    }
}
```

(Inicialmente i vale 1, hay que repetir mientras sea menor o igual que 10, y en cada paso hay que aumentar su valor una unidad),

De forma similar, podríamos hacer un contador de 1 a 5, como los que hemos creado con "while" y con "do-while":

```
// Contar1a5c.java
// Repetición con "for" 2: contar de 1 a 5
// Introducción a Java, Nacho Cabanes

class Contar1a5c {

    public static void main( String args[] ) {

        int x;

        for ( x=1 ; x<=5 ; x++ ) {
            System.out.println( x );
        }
    }
}
```

O bien podríamos contar descendiendo desde el 10 hasta el 2, con saltos de 2 unidades en 2 unidades, así:

```
// Contar10a0.java
// Repetición con "for" 3: descontar de 10 a 0, de 2 en 2
// Introducción a Java, Nacho Cabanes

class Contar10a0 {
```

```
    public static void main( String args[] ) {

        int i;

        for ( i=10 ; i>=0 ; i-=2 ) {
            System.out.println( i );
        }

    }
}
```

***Nota**: se puede observar una equivalencia casi inmediata entre la orden "for" y la orden "while". Así, el ejemplo anterior se podría reescribir empleando "while", de esta manera:*

```
// Contar10a0b.java
// Repetición con "while": descontar de 10 a 0, de 2 en 2
// Programa equivalente a "For2.java"
// Introducción a Java, Nacho Cabanes

class Contar10a0b {

    public static void main( String args[] ) {

        int i;

        i=10 ;
        while ( i>=0 ) {
            System.out.println( i );
            i-=2;
        }

    }
}
```

No hay obligación de declarar las variables justo al principio del programa. Es frecuente hacerlo buscando legibilidad, pero también se pueden declarar en puntos posteriores del fuente. Por eso,

un sitio habitual para **declarar la variable** que actúa de contador de un "for" es dentro del propio "for", así:

```
// ForVariable.java
// Variable declarada dentro de un "for"
// Introducción a Java, Nacho Cabanes

class ForVariable {

    public static void main( String args[] ) {

        for (int i=10 ; i>=0 ; i-=2 ) {
            System.out.println( i );
        }

    }
}
```

Esto tiene la ventaja de que no podemos reutilizar por error esa variable "i" después de salir del "for", porque "se destruye" automáticamente y obtendríamos un mensaje de error si tratamos de usarla. Así evitamos efectos indeseados de que cambios en una parte del programa afecten a otra parte del programa porque estemos reusando una variable sin darnos cuenta.

Ejercicio propuesto 6.3.1: Crea un programa que muestre los números del 1 al 15, usando "for"

Ejercicio propuesto 6.3.2: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "for"

Ejercicio propuesto 6.3.3: Crea un programa que muestre la "tabla de multiplicar del 5", usando "for"

Ejercicio propuesto 6.3.4: Crea un programa que muestre los números enteros del 0 al 100 que son múltiplos de 6 (el resto al dividir por 6 sea 0).

Ejercicio propuesto 6.3.5: Crea un programa que muestre los números enteros entre 0 y el que introduzca el usuario que sean múltiplos de 3 (el resto al dividir por 3 es 0) y a la vez múltiplos de 7 (ídem).

Ejercicio propuesto 6.3.6: Crea un programa que pida dos números de tipo byte al usuario y escriba en pantalla un rectángulo formado por asteriscos, que tendrá como alto el primer número y como ancho el segundo número.

Ejercicio propuesto 6.3.7: Crea un programa que pida un número de tipo byte al usuario y escriba en pantalla un cuadrado hueco de ese ancho, que tendrá un borde formado por asteriscos y su interior serán espacios en blanco. Por ejemplo, para un tamaño de 4 sería:

```
****
*  *
*  *
****
```

Ejercicio propuesto 6.3.8: Crea un programa que pida al usuario un número entero largo y muestre cuáles son sus divisores (aquellos entre los que se puede dividir, obteniendo como resto 0).

Ejercicio propuesto 6.3.9: Crea un programa que pida al usuario un número entero largo y diga si es primo (si sólo es divisible entre 1 y él mismo).

Ejercicio propuesto 6.3.10: Crea un programa que pida al usuario dos números enteros largos y diga cuántos números primos hay entre ellos (ambos incluidos).

Precaución con los bucles: Casi siempre, nos interesará que una parte de nuestro programa se repita varias veces (o muchas veces), pero no indefinidamente. Si planteamos mal la condición de salida, nuestro programa se puede quedar "colgado", repitiendo sin fin los mismos pasos (dentro de un "bucle sin fin").

break y continue

Se puede modificar un poco el comportamiento de estos bucles con las órdenes "break" y "continue".

La sentencia "**break**" hace que se salten las instrucciones del bucle que quedan por realizar, y se salga del bucle inmediatamente. Como ejemplo:

```
// Break1.java
// Ejemplo de uso de "break"
// Introducción a Java, Nacho Cabanes
```

```
class Break1 {

    public static void main( String args[] ) {

        int i;
        System.out.println( "Empezamos..." );
        for ( i=1 ; i<=10 ; i++ ) {
             System.out.println( "Comenzada la vuelta" );
             System.out.println( i );
             if (i==8)
                 break;
             System.out.println( "Terminada esta vuelta" );
        }
        System.out.println( "Terminado" );
    }
}
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, ni se darían las pasadas de i=9 e i=10, porque ya se ha abandonado el bucle.

La sentencia "**continue**" hace que se salten las instrucciones del bucle que quedan por realizar, pero no se sale del bucle sino que se pasa a la siguiente **iteración** (la siguiente "vuelta" o "pasada"). Como ejemplo:

```
// Continue1.java
// Ejemplo de uso de "continue"
// Introducción a Java, Nacho Cabanes

class Continue1 {

    public static void main( String args[] ) {

        int i;
        System.out.println( "Empezamos..." );
        for ( i=1 ; i<=10 ; i++ ) {
             System.out.println( "Comenzada la vuelta" );
             System.out.println( i );
```

```
                if (i==8)
                    continue;
                System.out.println( "Terminada esta vuelta" );
            }
            System.out.println( "Terminado" );
        }
    }
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, pero sí se darían la pasada de i=9 y la de i=10.

Ejercicio propuesto 6.4.1: Crea un programa que muestre los números del 1 al 20, excepto el 15, usando "for" y "continue"

Ejercicio propuesto 6.4.2: Crea un programa que muestre los números del 1 al 10, usando "for" que vaya del 1 al 20 y "break"

Sentencia return

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia ***return***. A diferencia de ***continue*** o ***break***, la sentencia ***return*** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (return value;).

Contacto con las funciones.

Descomposición modular

En ocasiones, nuestros programas contendrán operaciones repetitivas. Crear funciones nos ayudará a que dichos programas sean más fáciles de crear y más robustos (con menos errores). Vamos a verlo con un ejemplo...

Imaginemos que queremos calcular la longitud de una circunferencia a partir de su radio, y escribirla en pantalla con dos cifras decimales. No es difícil: por una parte, la longitud de una circunferencia se calcula con 2 * PI * radio; por otra parte, para conservar sólo dos cifras decimales, lo podemos hacer de varias formas, una de las cuales consiste en multiplicar por 100, quedarnos con la parte entera del número y volver a dividir entre 100:

```
// FuncionesPrevio1.java
```

```
// Ejemplo previo 1 de la conveniencia de usar funciones
// para evitar código repetitivo

import java.io.*;

class FuncionesPrevio1 {
    public static void main( String args[] ) {

        int radio = 4;

        double longCircunf = 2 * 3.1415926535 * radio;
        double longConDosDecimales =
            Math.round(longCircunf * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio + " es " + longConDosDecimales);
    }
}
```

Su resultado sería

```
La longitud de la circunferencia de radio 4 es 25.13
```

Si ahora queremos hacerlo para 5 circunferencias, podemos repetir esa misma estructura varias veces:

```
// FuncionesPrevio1.java
// Ejemplo previo 2 de la conveniencia de usar funciones
// para evitar código repetitivo

import java.io.*;

class FuncionesPrevio2 {
    public static void main( String args[] ) {

        int radio1 = 4;

        double longCircunf1 = 2 * 3.1415926535 * radio1;
        double longConDosDecimales1 =
```

```
            Math.round(longCircunf1 * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio1 + " es " + longConDosDecimales1);

        int radio2 = 6;

        double longCircunf2 = 2 * 3.1415926535 * radio2;
        double longConDosDecimales2 =
            Math.round(longCircunf2 * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio2 + " es " + longConDosDecimales2);

        int radio3 = 8;

        double longCircunf3 = 2 * 3.1415926535 * radio3;
        double longConDosDecimales3 =
            Math.round(longCircunf3 * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio3 + " es " + longConDosDecimales3);

        int radio4 = 10;

        double longCircunf4 = 2 * 3.1415926535 * radio4;
        double longConDosDecimales4 =
            Math.round(longCircunf4 * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio4 + " es " + longConDosDecimales4);

        int radio5 = 111;

        double longCircunf5 = 2 * 3.1415926535 * radio5;
        double longConDosDecimales5 =
            Math.round(longCircunf5 * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio5 + " es " + longConDosDecimales5);
    }
}
```

Pero un programa tan repetitivo es muy propenso a errores: tanto si escribimos todo varias veces como si copiamos y pegamos, es fácil que nos equivoquemos en alguna de las operaciones, usando el nombre de una variable que no es la que debería ser. Por ejemplo, podría ocurrir que escribiéramos "double longCircunf5 = 2 * 3.1415926535 * radio4;" (en el último fragmento no aparece "radio5" sino "radio4"). El programa se comportaría de forma incorrecta, y este error podría ser muy difícil de descubrir.

La alternativa es crear una "función": un bloque de programa que tiene un nombre, que recibe ciertos datos, y que puede incluso devolvernos un resultado. Por ejemplo, para el programa anterior, podríamos crear una función llamada "escribirLongCircunf" (escribir la longitud de la circunferencia), que recibiría un dato (el radio, que será un número entero) y dará todos los pasos que daba nuestro primer "main":

```
public static void escribirLongCircunf( int radio ) {
    double longCircunf = 2 * 3.1415926535 * radio;
    double longConDosDecimales =
        Math.round(longCircunf * 100) / 100.0;
    System.out.println( "La longitud de la circunferencia " +
        "de radio " + radio + " es " + longConDosDecimales);
}
```

Y desde "main" (el cuerpo del programa) usaríamos esa función tantas veces como quisiéramos:

```
public static void main( String args[] ) {
    escribirLongCircunf(4);
    escribirLongCircunf(6);
    ...
```

De modo que el programa completo quedaría:

```
// Funciones01.java
// Primer ejemplo de funciones

import java.io.*;

class Funciones01 {

    public static void escribirLongCircunf( int radio ) {
```

```
        double longCircunf = 2 * 3.1415926535 * radio;
        double longConDosDecimales =
            Math.round(longCircunf * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio + " es " + longConDosDecimales);
    }

    public static void main( String args[] ) {
        escribirLongCircunf(4);
        escribirLongCircunf(6);
        escribirLongCircunf(8);
        escribirLongCircunf(10);
        escribirLongCircunf(111);
    }
}
```

Por ahora, hasta que sepamos un poco más, daremos por sentado que todas las funciones tendrán que ser "**public**" y "**static**".

Ejercicio propuesto 9.1.1: Crea una función llamada "borrarPantalla", que borre la pantalla dibujando 25 líneas en blanco. No debe devolver ningún valor. Crea también un "main" que permita probarla.

Ejercicio propuesto 9.1.2: Crea una función llamada "dibujarCuadrado3x3", que dibuje un cuadrado formato por 3 filas con 3 asteriscos cada una. Crea también un "main" para comprobar que funciona correctamente.

Parámetros

Nuestra función "escribirLongCircunf" recibía un dato entre paréntesis, el radio de la circunferencia. Estos datos adicionales se llaman "**parámetros**", y pueden ser varios, cada uno indicado con su tipo de datos y su nombre.

Ejercicio propuesto 9.2.1: Crea una función que dibuje en pantalla un cuadrado del ancho (y alto) que se indique como parámetro. Completa el programa con un "main" que permita probarla.

Ejercicio propuesto 9.2.2: Crea una función que dibuje en pantalla un rectángulo del ancho y alto que se indiquen como parámetros. Completa el programa con un "main" que permita probarla.

Valor de retorno

Las funciones "**void**", como nuestra "escribirLongCircunf" y como el propio cuerpo del programa ("main") son funciones que dan una serie de pasos y no devuelven ningún resultado. Este tipo de funciones se suelen llamar "**procedimientos**" o "**subrutinas**".

```
public static void saludar( ) {
    System.out.println( "Bienvenido");
    System.out.println( "Comenzamos...");
}
```

Por el contrario, las funciones matemáticas suelen dar una serie de pasos y devolver un resultado, por lo que no serán "void", sino "**int**", "double" o del tipo que corresponda al dato que devuelven. Por ejemplo, podríamos calcular la superficie de un círculo así:

```
public static double superfCirculo( int radio ) {
    double superf = 3.1415926535 * radio * radio;
    double superfConDosDecimales =
        Math.round(superf * 100) / 100.0;
    return superfConDosDecimales;
}
```

O descomponer la parte matemática del ejemplo anterior (el cálculo de la longitud de la circunferencia) en una función independiente, así:

```
public static double longCircunf( int radio ) {
    double longitud = 2 * 3.1415926535 * radio;
    double longConDosDecimales =
        Math.round(longitud * 100) / 100.0;
    return longConDosDecimales;
}
```

Y un programa completo quedaría que usara todas esas funciones quedaría:

```
// Funciones02.java
// Segundo ejemplo de funciones

import java.io.*;

class Funciones02 {
```

```
    public static double longCircunf( int radio ) {
        double longitud = 2 * 3.1415926535 * radio;
        double longConDosDecimales =
            Math.round(longitud * 100) / 100.0;
        return longConDosDecimales;
    }

    public static double superfCirculo( int radio ) {
        double superf = 3.1415926535 * radio * radio;
        double superfConDosDecimales =
            Math.round(superf * 100) / 100.0;
        return superfConDosDecimales;
    }

    public static void saludar( ) {
        System.out.println( "Bienvenido");
        System.out.println( "Comenzamos...");
    }

    public static void escribirLongCircunf( int radio ) {
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio + " es " + longCircunf( radio ));
    }

    public static void main( String args[] ) {
        saludar();
        escribirLongCircunf(4);
        escribirLongCircunf(6);
        escribirLongCircunf(8);
        escribirLongCircunf(10);
        escribirLongCircunf(111);
        System.out.println( "La superficie del círculo " +
            "de radio 5 es " + superfCirculo(5));
    }
}
```

Volveremos más adelante a las funciones, para formalizar un poco lo que hemos aprendido, y también para ampliarlo, pero ya tenemos suficiente para empezar a practicar.

Ejercicio propuesto 9.3.1: Crea una función que calcule el cubo de un número real (float) que se indique como parámetro. El resultado deberá ser otro número real. Probar esta función para calcular el cubo de 3.2 y el de 5.

Ejercicio propuesto 9.3.2: Crea una función que calcule el menor de dos números enteros que recibirá como parámetros. El resultado será otro número entero.

Ejercicio propuesto 9.3.3: Crea una función que devuelva la primera letra de una cadena de texto. Probar esta función para calcular la primera letra de la frase "Hola".

Ejercicio propuesto 9.3.4: Crea una función que devuelva la última letra de una cadena de texto. Probar esta función para calcular la última letra de la frase "Hola".

Ejercicio propuesto 9.3.5: Crea una función "esPrimo", que reciba un número y devuelva el valor booleano "true" si es un número primo o "false" en caso contrario.

Calendar

La clase Calendar posee una gran cantidad de métodos para operar, consultar y modificar las propiedades de una fecha. Un aspecto principal es que es una clase abstracta y como tal posee algunos métodos que deben ser implementados por sus subclases.

Calendar se suele instanciar con su método getInstance() el cual nos crea un objeto de la clase conteniendo la fecha de ese momento. Así es muy típico el uso: **Calendar** ahoramismo = **Calendar**.getInstance();
El conjunto de métodos “set” permite establecer una fecha, mientras que los métodos “add” y “roll” permiten cambiar las fechas sumando o restando una cantidad. Estos dos últimos métodos fuerzan que los valores para los campos no sobrepasen el mínimo o el máximo del permitido según el calendario. También estos métodos suponen un recálculo inmediato de la fecha tras el cambio de sus valores, cosa que no ocurre con el uso de los métodos set.
Por ejemplo imaginemos que tenemos una fecha con los datos siguientes: 31 de Agosto de 2000. Si ahora llamáramos sobre ese objeto Calendar el método add(Calendar.MONTH, 13) lo que hacemos es añadir 13 meses más a la fecha, con lo que tendríamos la fecha 30 de Septiembre del 2001. Observamos que al añadir 13 meses nos vamos al mes siguiente del año siguiente, pero sin saltar ninguna excepción hemos pasado del día 31 al 30, esto es porque add como hemos

dicho fuerza los valores para que no sobrepasen del mínimo o máximo del campo correspondiente, en este caso del campo de días.

Funciones Clase Math Java

En cuanto a las funciones matemáticas en Java, las funciones disponibles vienen definidas en la clase Math. Hay muchas funciones disponibles.
A continuación, mostraremos las funciones más importantes y ejemplos de uso:

Función matemática	**Significado**	**Ejemplo de uso**	**Resultado**
abs	Valor absoluto	int x = Math.abs(2.3);	x = 2;
atan	Arcotangente	double x = Math.atan(1);	x = 0.78539816339744;
sin	Seno	double x = Math.sin(0.5);	x = 0.4794255386042;
cos	Coseno	double x =	x =

		Math.cos(0.5);	0.87758256189037;
tan	Tangente	double x = Math.tan(0.5);	x = 0.54630248984379;
exp	Exponenciación neperiana	double x = Math.exp(1);	x = 2.71828182845904;
log	Logaritmo neperiano	double x = Math.log(2.7172);	x = 0.99960193833500;
pow	Potencia	double x = Math.pow(2.3);	x = 8.0;
round	Redondeo	double x = Math.round(2.5);	x = 3;
random	Número aleatorio	double x = Math.ramdom();	x = 0.20614522323378;
floor	Redondeo al entero menor	double x = Math.floor(2.5);	x = 2.0;
ceil	Redondeo al entero mayor	double x = Math.ceil(2.5);	x = 3.0;

Destacar que las funciones matemáticas, al pertenecer a la clase Math, se invocan siempre de la siguiente manera: Math.funcion(argumentos).

Las funciones relacionadas con ángulos (**atan, cos, sin, tan**, etc.) trabajan en radianes. Por tanto, para operar con grados, tendremos que realizar la conversión oportuna. La propia clase Math facilita los métodos toRadians para transformar grados sexagesimales en radianes y toDegrees para transformar radianes en grados sexagesimales, aunque las conversiones pueden no ser

totalmente precisas. Por ejemplo cos(toRadians(90.0)) debería devolver 0, pero es probable que devuelva un valor aproximadamente cero pero no exactamente cero debido a que la precisión decimal no es absoluta.
La función **random**,permite generar números aleatorios en el rango]0,1[. Por tanto el 0 y el 1 están excluidos.
La función exponenciación neperiana o exponenciación de e, matemáticamente significa **ex**, que en Java sería **Math.exp(x)**,donde xes un número real y la base es la constante neperiana e = 2.7172...
La función logaritmo neperiano, matemáticamente significa **Ln x**, que en Java correspondería a la expresión **Math.log(x).**
La función potencia, matemáticamente significa **baseexponente**, que en Java se convertiría en **Math.pow(base,exponente),**donde base y exponente son números reales, por lo tanto, si queremos obtener la raíz cubica de 2,la instrucción sería Math.pow(2,0.333).

CALCULAR EL TIEMPO DE EJECUCION DE UN PROGRAMA JAVA

Muchas veces es necesario desarrollar un programa eficiente, por ello es importante calcular el tiempo de ejecución de un programa en Java. Esto se calcula mediante la función **System.currentTimeMillis()** que nos da el tiempo actual en milisegundos.
Así podríamos calcular el tiemp o en un momento dado, luego realizar las funciones que necesitemos medir y finalmente volver a calcular el tiempo en ese nuevo momento para restarlo al anterior y obtener el tiempo que tardaron las funciones intermedias.

Veamos un ejemplo completo de ello:

```
public class pruebaTiempo {
    public static void main(String args[]){
        long totalTiempo;
        long tiempoInicio;

        // Inicializa con el tiempo actual
        tiempoInicio = System.currentTimeMillis();

        // Funciones a medir, como ejemplo un bucle for imprimiendo por
pantalla un hola mundo
        for(int i=0 ; i&lt;10 ; i++)
            System.out.println("¡ Hola mundo !");

        // Resta el tiempo de inicio al actual resultando el tiempo usado
        totalTiempo = System.currentTimeMillis() - tiempoInicio;

        System.out.println("Tiempo demorado:\t" + totalTiempo + "
```

```
milisegundos.");
     }
}
```

Muy simple. Para cambiar la unidad simplemente multiplicando por 1000 obtendríamos en segundos y por 60000 en minutos y así sucesivamente.

A veces queremos medir eficiencias mas cortas que milisegundos, y no nos valdría con simplemente cambiar la unidad dado que la función **System.currentTimeMillis()** mide como mínimo en milisegundos. Entonces que podríamos hacer…

En Java SE 5 se implemento un nuevo método para System: **System.nanoTime()**, que nos dará el resultado en nanosegundos. Con nanosegundos podremos medir la eficiencia con mucha mas precisión y en funciones mas cortos, por ejemplo, inserciones únicas en una base de datos.

Veamos un ejemplo con nanoTime():

```
public class pruebaTiempo {
     public static void main(String args[]){
          long totalTiempo;
          long tiempoInicio;

          // Inicializa con el tiempo actual
          tiempoInicio = System.nanoTime();

          // Funciones a medir, como ejemplo un bucle for imprimiendo por
pantalla un hola mundo
          for(int i=0 ; i<10 ; i++)
               System.out.println("¡ Hola mundo !");

          // Resta el tiempo de inicio al actual resultando el tiempo usado
          totalTiempo = System.nanoTime() - tiempoInicio;

          System.out.println("Tiempo demorado:\t" + totalTiempo + "
nanosegundos.");
     }
}
```

Bibliografía:

www.aprendeaprogramar.com
Introducción a Java

JAVA – Programación Orientada a Objetos
Andres Juarez
Editorial Eudeba

Aprenda Java como si estuviera en Primero
Javier García de Jalón, José Ignacio Rodríguez
Iñigo Mingo, Aitor Imaz, Alfonso Brazález, Alberto Larzabal , Jesús Calleja, Jon García
Escuela Superior de Ingenieros Industriales
UNIVERSIDAD DE NAVARRA

JAVA 2 – Curso de Programación 4ta Edición
Francisco Javier Ceballos
Editorial Alfaomega

Piensa en JAVA 4ta Edición
Bruce Eckel
Editorial Pearson

JAVA 7
Hebert Schildt
Editorial Amaya